Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Desarrollo e investigación de aplicaciones software formativas en enseñanza superior con el motor gráfico Unreal Engine

Autor: Sara Mayoral Marín

Tutora: Elisa Rojas Sánchez

2021

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Desarrollo e investigación de aplicaciones software formativas en enseñanza superior con el motor gráfico Unreal Engine

Autora: Sara Mayoral Marín

Tutora: Elisa Rojas Sánchez

Tribunal:

Presidente: Miguel Ángel López Carmona

Vocal 1º: Isaías Martínez Yelmo

Vocal 2º: Elisa Rojas Sánchez

Fecha de depósito: 2 de septiembre de 2021

Agradecimientos

Este trabajo es el fruto de muchas horas de trabajo, a lo largo de las cuales ha habido tantos altibajos que ya he perdido la cuenta. Quiero agradecer a todos aquellos que han estado ahí conmigo. A mis padres por la paciencia durante estos años y a mi hermana por estar ahí siempre que la he necesitado. También agradecer a Javier por darme la sugerencia de hablar Elisa para realizar este trabajo. Gracias también a Elisa por su ayuda, ideas y por sus refrescantes clases, que han hecho la enseñanza durante la pandemia un poco más llevadera. Por último, gracias a mi pareja por el apoyo emocional durante estos años.

Resumen

A lo largo de este documento se va a hablar sobre el motor gráfico Unreal Engine y las posibles aplicaciones para la enseñanza superior. Se hará un breve resumen de los orígenes de los motores gráficos para videojuegos y se compararán el motor Unreal Engine con el motor Unity, explicando el porqué de la elección del motor. A continuación, se dará una breve explicación de los conceptos básicos de Unreal, así como los primeros pasos para poder trabajar con el motor. Expuesto lo anterior se pasará a explicar el diseño y la implementación del proyecto para demostrar la aplicabilidad del motor en la enseñanza superior. Por último se hablará de los resultados obtenidos, las conclusiones y el trabajo futuro.

Palabras clave: Motores Gráficos, Unreal Engine, Desarrollo de aplicaciones en la enseñanza.

Abstract

Throughout this paper we will discuss the Unreal Engine graphics engine and possible applications for higher education. A brief summary of the origins of graphics engines for video games will be made and the Unreal Engine will be compared with the Unity engine, explaining why this engine was chosen. Next, a brief explanation of the basic concepts of Unreal will be given, as well as the first steps to be able to work with the engine. After that, the design and implementation of the project will be explained in order to demonstrate the applicability of this engine in higher education. Finally, the results obtained, conclusions and future work will be discussed.

Keywords: Unreal Engine, graphics engine, development of teaching applications.

Índice general

Capítulo 1

Introducción

La pandemia acontecida este último año ha obligado al mundo a modificar drásticamente su forma de vida. Determinados trabajos, aquellos no imprescindibles, han tenido que adaptarse a una situación que se ha visto obligada a limitar el contacto humano, haciendo uso del tele-trabajo. El sector de la educación, tanto la infantil, media y superior han tenido que adaptarse a estos cambios haciendo uso de plataformas que permitan dar clases online.

A raíz de esta situación se ha potenciado una tendencia dentro de la enseñanza: la gamificación. Esta técnica traslada la mecánica de los juegos al ámbito educativo-profesional. El profesor intenta crear un ambiente de dinamismo en la clase y que aumente la participación en el aula.

Para cumplir estos objetivos se pueden usar diferentes metodologías [1], como pueden ser un sistema de puntos y recompensas, o el uso de herramientas más complejas, como la aplicación de Kahoot [2]. Otro ejemplo, es el uso de videojuegos diseñados específicamente para estos ámbitos, como es el caso de el Minecraft que tiene una versión especialmente diseñada para la educación en colegios e institutos [3].

Una de las principales ventajas que encontramos en esta metodología es que consigue que el entorno dentro del aula sea más interactivo, ya sea entre el alumno y el profesor o entre los propios alumnos. Además, el uso de estas herramientas ayuda a la motivación por parte de los estudiantes y a una personalización en las actividades por parte de los profesores lo que ayuda a una mejor comprensión y adquisición de los conocimientos impartidos [4].

1.1 Objetivos

Teniendo en cuenta lo mencionado anteriormente, el objetivo principal de este trabajo de fin de grado (TFG) es investigar y, usando los conocimientos adquiridos, desarrollar un caso de uso de aplicación del motor gráfico Unreal Engine 4 en una actividad formativa en educación superior, orientada principalmente al refuerzo de los conocimientos adquiridos en el aula.

Como se ha comentado antes, la enseñanza superior se enfrenta al reto de la docencia online por lo que otra de las motivaciones para realizar este proyecto es investigar nuevas metodologías para la enseñanza universitaria, haciendo uso de nuevas tecnologías, aplicando el concepto de la gamificación y el uso de realidad aumentada y virtual. Enfocando así estas tecnologías a un estudio más interactivo y dinámico de los conocimientos impartidos o como soporte a la enseñanza más clásica.

Este trabajo trata de averiguar el alcance técnico, así como el presupuesto y el tiempo requerido para realizar un proyecto de estas características.

1.2 Campos de aplicación

La aplicación más evidente del empleo de motores gráficos en la enseñanza es el uso de aplicaciones que ayuden al docente a generar un ambiente más dinámico dentro del aula, así como el estudio y refuerzo de los conocimientos adquiridos por parte de los alumnos. Este uso va a ser el punto central de este trabajo.

A pesar de ello, al tratarse de un estudio dirigido a la enseñanza superior, otra posible aplicación sería el propio uso de estas herramientas por parte de los alumnos, con la ayuda orientativa por parte del profesorado, es decir, que sean los propios alumnos los que desarrollen diferentes aplicaciones.

1.3 Estructura de la memoria

A lo largo de este documento se va a explicar los conceptos básicos de los motores gráficos para videojuegos. Se hablará del motivo y los orígenes de su creación. Se explicará brevemente los dos grandes motores que se usan en la industria, explicando sus diferencias y puntos en común, así como las herramientas complementarias que se usan juntos a ellos.

Se explicará más en detalle la instalación y creación de proyectos en Unreal, así como conceptos básicos y consideraciones a la hora de diseñar y crear un videojuego con este motor.

A continuación, se hablará del proyecto creado en base a estos conocimientos, relacionándolos con la gamificación y con la formación universitaria. Se explicará el diseño del proyecto, así como la implementación y el desarrollo del mismo dentro del motor.

Por último, se hablará de los resultados obtenidos en base al proyecto, dificultades encontradas y posibles mejoras, así como futuros proyectos relacionados con este trabajo de fin de grado y con los motores gráficos.

Capítulo 2

Motores gráficos para videojuegos

2.1 Qué es un motor gráfico de videojuegos

Conceptualmente hablando, un motor gráfico de videojuegos, o simplemente un motor gráfico, es un entorno de desarrollo software necesario para la creación de videojuegos [5]. Estos motores son diferentes a los entornos de desarrollo tradicionales y son, por lo general, complementarios.

Estos motores se encargan de construir juegos para diferentes plataformas tan diversas como dispositivos móviles, consolas u ordenadores personales. Las funcionalidades principales que proporcionan los motores suelen incluir núcleo de renderizado tanto para gráficos 2D y 3D, un motor de físicas o de colisiones, así como diferentes funcionalidades para la gestión de sonido, vídeos, animaciones, módulos de inteligencia artificial y gestión de redes. Así mismo, proporciona herramientas para la gestión de la memoria e hilos [6].

Estas herramientas de desarrollo visual, así como los componentes software son reutilizables, al proporcionarse en un entorno integrado. Esto permite un desarrollo rápido y simplificado, permitiendo que los desarrolladores se centren en la información que quieren transmitir.

Este tipo de motores se suelen denominar *middleware* [7], ya que desde un punto de vista empresarial son una plataforma software que proporciona las bases de la funcionalidad y que permiten el desarrollo de un juego reduciendo sus costes y el tiempo de desarrollo, algo crítico para la industria. Al igual que otros tipos de plataforma *middleware*, los motores gráficos proporcionan un alto nivel de abstracción, esto permite que el mismo juego se desarrolle para diferentes plataformas con cambios mínimos en el código fuente. En algunos casos los motores se diseñan en base a componentes de tal manera que se puedan combinar de forma selectiva diseñando un motor personalizado y especifico para el juego a desarrollar. Esto además proporciona una mayor escalabilidad [8].

A pesar del concepto general que se tiene de ellos, este tipo de motores no se usan exclusivamente para la industria del videojuego y a lo largo de los años se ha ido ampliando su mercado a usuarios finales que reutilizan las capacidades del motor para otros tipos de aplicaciones interactivas, como demostraciones de marketing, simulaciones de partículas e incluso visualizaciones arquitectónicas

2.2 Historia de los motores gráficos

2.2.1 Inicios

Antes de la creación de los motores gráficos, los videojuegos se codificaban como entidades independientes [9] y el diseño arquitectónico del programa era clave a la hora de optimizarlo. Por ejemplo, para el sistema Atari 2600 (ver Figura 2.1) los juegos se diseñaban con un modelo bottom-up[1] que permitía hacer un uso más optimizado de los recursos del sistema.

Figura 2.1: Imagen de la Atari 2600 [10]

Otras plataformas tenían un mayor margen de maniobra en cuanto a la optimización de los gráficos. Sin embargo, la mayoría de las plataformas compartían la preocupación común de las limitaciones de memoria. Lo cual impedía y limitaba el desarrollo de aplicaciones con un diseño de datos pesados, ya que estos requerían de una gestión especial de la memoria. Estas optimizaciones eran poco reutilizables entre juegos.

2.2.2 Primeros prototipos

A lo largo de los años 80, surgió una ola de popularidad en los videojuegos, la consola Atari tuvo que competir con las empresas japonesas de Nintendo y SEGA, estas consolas tenían una mayor potencia de computación y memoria [11]. Paralelamente, los ordenadores de sobremesa comenzaron a comercializarse al público general con precios más asequibles estableciéndose así de forma permanente la industria de los videojuegos.

Durante esta época, las empresas seguían teniendo un desarrollo independiente para cada juego y los motores de terceras empresas todavía no estaban establecidos en el mercado. Sin embargo, surgieron varios sistemas de creación de juegos 2D que facilitaban el desarrollo independiente para juegos de ordenador.

Uno de estos sistemas fue el Pinball Construction Set(ver Figura 2.2), creado por Bill Budge para el Apple II, este juego permitía al jugador diseñar sus propios mapas de pinball, incluía atributos que permitían modificar la gravedad, cambiar el fondo y el estilo del nivel entre otras cosas. El juego alcanzo mucha popularidad y se llegaron a sacar versiones para IBM y para el Macintosh en 1986 [12].

2.2.3 El auge de los gráficos 3D

Ya a finales de los años 80 y principios de los 90, los videojuegos dieron un salto enorme cuando se paso a la tecnología de 16-bits, gracias a la SNES y Mega Drive (ver Figura 2.3). Esta nueva tecnología trajo consigo importantes mejoras gráficas.

Durante estos años nació la primera PlayStation (1994) de la empresa Sony. A pesar de las mejoras evidentes en las consolas especificas el verdaderos protagonista fue el PC. En él comienzan a aparecer

[1] Un modelo de programación Bottom-up es un estilo de programación donde la aplicación se construye empezando con primitivas básicas del lenguaje de programación y, gradualmente, añadiendo características más complejas.

Figura 2.2: Pantalla inicial de Pinball Construction Set [13]

(a) SNES [14] (b) Mega drive [15]

Figura 2.3: Primeras consolas basadas en 16-bits

juegos cada vez más avanzados tecnológicamente, nacen los shooters 3D como DOOM (ver Figura 2.4) y Quake.

Debido a la popularidad y la complejidad técnica que requería el desarrollo, los desarrolladores de otros juegos adquirieron la licencia de algunas partes de la lógica del juego, permitiéndoles centrarse en el diseño de los gráficos, personajes, armas y niveles; lo que en su momento se denominó los recursos del juego. Esto permitió no tener que trabajar desde cero y poder empezar el desarrollo con una base sólida y funcional [11]. Esta diferenciación entre las reglas o lógica del juego, los datos específicos y los conceptos básicos como la detección de colisiones permitió que los equipos pudiesen crecer y especializarse.

Figura 2.4: Captura del juego Doom (1995) [16]

A medida que la industria crecía, los juegos que se iban desarrollando adoptaron este enfoque: la lógica de comportamiento, es decir, el motor, por un lado y el contenido del juego por otro. Esto permitía que una empresa con una gran capacidad de desarrollo como Epic Games [17] o id Software [18], otorgasen licencias de sus motores como fuente de ingresos adicional. Otra ventaja de este modelo de negocio es que el desarrollo de videojuegos se abarató en costes, se redujo el tiempo y se facilitó el desarrollo.

2.2.4 Uso de motores gráficos en la actualidad

Durante estos últimos años, la industria del videojuego ha conseguido superar en beneficios económicos a la industria musical y cinematográfica. Como consecuencia de ello, han salido al mercado diversas herramientas, tanto gratuitas como de pago, que permiten el desarrollo y la creación de forma más o menos sencilla de este tipo de contenido.

Actualmente, los motores gráficos son una de las aplicaciones software más complejas del mercado, en muchas ocasiones con docenas de sistemas que interactúan entre si. La evolución en el desarrollo de videojuegos ha provocado una fuerte diferenciación entre el renderizado, el *scripting*, los materiales gráficos y el propio diseño del juego. Es muy común, en los equipos de desarrollo, que el número de programadores de código se vea superado por el numero de artistas, el principal motivo de ello son los motores gráficos, ya que proporcionan toda la lógica básica de desarrollo liberando gran parte de carga de los programadores.

Además, la gran cantidad de documentación y tutoriales permite el estudio autodidacta facilitando que no solo alguien especializado en el desarrollo software pueda desarrollar y terminar un producto final.

Los dos motores gráficos sobre los que oscilan los desarrolladores mas independientes son el motor Unreal Engine, que actualmente va por su versión 5 y el motor Unity cuya versión es la 2021.1.12. Más adelante se explicarán las principales diferencias entre ambos y cual se ha seleccionado para el presente TFG.

2.3 Estándar de la industria

La elección de un motor gráfico es un tema de debate incluso entre los desarrolladores más experimentados. Por lo general, las grandes empresas como Rockstar [19] o Ubisoft [20], que poseen un mayor presupuesto, son capaces de desarrollar su propio motor, adaptándose a las necesidades del juego que van a desarrollar en ese momento; ahorrándose los problemas de licencia que suele haber cuando se desarrolla algo con herramientas no propias.

Si nos alejamos de las grandes corporaciones y nos vamos a un ámbito mas *indie*, los dos motores mas habituales suelen ser en Unreal Engine 4 y Unity [21]. Esta elección es principalmente debido a que ambos poseen licencia libre y tienen la capacidad y calidad suficiente para crear y desarrollar un videojuego sin demasiados problemas por parte de la herramienta, además, ambos motores poseen una gran comunidad con tutoriales tanto de pago como gratuitos y las empresas desarrolladoras actualizan regularmente el motor en función de las necesidades del mercado.

2.4 Herramientas complementarias

Hasta el momento, nos hemos centrado principalmente en el motor gráfico como tal, sus características principales, usos dentro del desarrollo,etc. Sin embargo, es muy difícil que con solo un motor gráfico seamos capaces de desarrollar un videojuego. Los modelos en 3D, las animaciones, la música, los efectos, texturas y materiales entre muchos otros son los denominados *assets*. Para la creación de estos elementos se usan diferentes programas para generarlos y aunque no se van a explicar en profundidad, conviene comentarlos y tenerlos en cuenta porque juegan un papel vital en el desarrollo. No solo hacen posible la creación del producto, si no que un mal uso de ellos o una elección incorrecta puede llegar incluso a afectar al rendimiento del producto final.

Hay varias formas de obtener un *asset* en función de lo que se vaya buscando. Si se posee un gran presupuesto, la mejor opción suele ser contratar profesionales de cada campo que diseñen y creen los *assets* específicos para el proyecto. Esta opción, aunque es la ideal, ya que se posee un mayor control sobre la calidad final de los *assets*, no siempre es posible. Con un bajo presupuesto, las opciones más viables es la compra de *assets* ya creados, ya sea por la comunidad o por artistas independientes, esto reduce el coste de producción y el tiempo de desarrollo, pero puede provocar que la cohesión visual del resultado final no sea la esperada.

Para la creación de cada *asset* se usan diferentes herramientas y programas. Por ejemplo, para la creación de modelos en 3D se usa Autodesk Maya [22] o su opción gratuita Blender [23], para los efectos sonoros pasa algo parecido, hay muchas páginas y bibliotecas de sonidos tanto de pago como gratuitas. A lo largo de este trabajo se van a explicar de dónde se han obtenido los *assets* usados, pero no se va a profundizar demasiado ya que se aleja mucho de nuestro campo de estudio.

2.5 Qué motor gráfico elegir

Como se ha mencionado anteriormente, la elección de un motor gráfico es algo muy a tener en cuenta a la hora de desarrollar el producto final, dentro del desarrollo *indie* las dos opciones más habituales son Unreal y Unity [21]. Esta elección del motor debe hacerse al principio del desarrollo, pues es la que determina el flujo a seguir, cada motor tiene sus propias características, por ejemplo, el tipo de lenguaje o los módulos que vienen implementados en el propio motor de forma nativa. A lo largo de este apartado se va a explicar las principales diferencias entre los dos motores, Unity y Unreal, y los motivos que han llevado a elección final para este trabajo.

2.5.1 Características generales

El motor gráfico Unreal se publicó en el año 1998, por parte de la compañía Epic Games[17]. Esta empresa de videojuegos dedicó sus esfuerzos en refinar y pulir las herramientas del motor y el soporte del mismo, para poder sacar un beneficio adicional con la venta de licencias del motor. Esta licencia era muy cara, llegando incluso a precios que rondaban las cinco cifras, pero esta política ha ido cambiando a lo largo de los años y el pago de licencia es muy diferente a lo que era en la década de los 2000 (de esto se hablará más adelante). Gracias a este motor se han podido crear juegos como la saga Bioshock [24] (ver Figura 2.5) o la saga Deus Ex [25]. Por último, este motor, al tratarse de un motor originalmente pensado para ordenador, no tiene demasiado soporte para móvil[26].

(a) Bioshock (2007) [27] (b) Bioshock infinite(2013) [28]

Figura 2.5: Saga Bioshock

Por otro lado, en motor Unity nació en el año 2005, por la empresa desarrolladora Unity Technologies [29]. Este motor nació originalmente como motor exclusivo para la arquitectura IOS de Mac, hasta el año

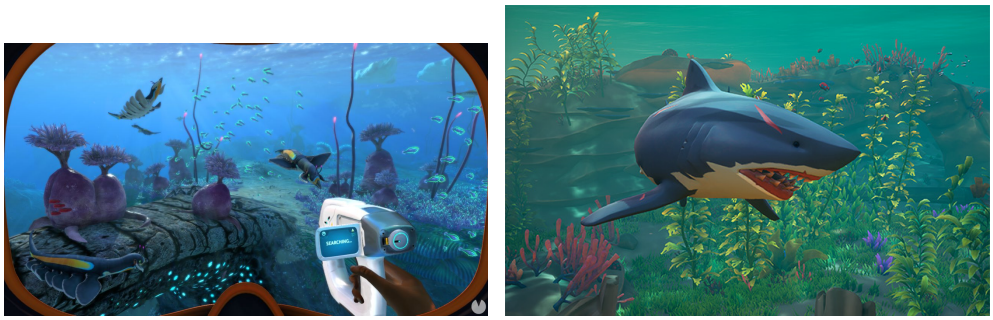
2007, en el cual la empresa decidió añadir nuevas características y ampliar las plataformas disponibles. A diferencia de Unreal, este motor se ha dirigido a un desarrollo mas indie, y el número de plataformas objetivo es sustancialmente más alto. Hay que puntualizar que, a lo largo de estos años, esta diferencia se ha reducido considerablemente. También es importante comentar que, durante estos últimos años, Unity ha ido desarrollando y añadiendo nuevas características centradas en el rendimiento y tecnologías de renderizado de gráficos, acercándose peligrosamente a la calidad de gráficos de Unreal, sin llegar a superarlo.

2.5.2 Gráficos

Una de las cosas a tener en cuenta es la calidad de los gráficos y la gestión que hace el motor sobre ellos. Actualmente ambos motores poseen una gran capacidad para general gráficos de alta calidad. A pesar de ello, hay ciertas consideraciones a valorar. Tal y como se ha comentado anteriormente, la diferencia de gráficos es mucho más pequeña cada año, llegando incluso a ser indistinguible [30].

Otra consideración es que si el objetivo del desarrollo es alcanzar gráficos hiperrealistas, la mejor opción es Unreal debido a que, a pesar de que Unity ha mejorado la calidad de los gráficos, su rendimiento y resultado final en este tipo de gráficos no llegan a la calidad de Unreal. Por lo que si lo que se quiere es un juego cuyo atractivo sean una calidad foto realista, Unreal es la opción adecuada, ya que posee herramientas que permiten gestionar grandes sistemas de partículas, geometrías con una gran carga de poligonaje, mayor número de luces, *shaders* tanto simples como complejos, entre muchos otros[30].

Esto no significa que un juego diseñado en un entorno Unreal tenga automáticamente mejores gráficos que uno en Unity, pero la gestión y optimización que hace el Unreal es mucho mejor que Unity. Además, el trabajo requerido para alcanzar la misma calidad de gráficos es mucho menor en Unreal que en Unity. En las imágenes de la Figura 2.6 se pueden ver dos juegos con unos gráficos estilizados de los dos motores. Como se observa se pueden obtener resultados similares.



(a) Subnautica (Unity) [31] (b) Sea of Thieves (Unreal Engine) [32]

Figura 2.6: Diferencia de gráficos estilizados en los dos motores

2.5.3 Curva de aprendizaje y experiencia del programador

Respecto a la curva de aprendizaje, si no se tiene ninguna experiencia con motores gráficos o incluso con herramientas gráficas, Unreal puede llegar a ser abrumador. La interfaz de usuario es compleja con muchas pestañas y paneles. Unreal permite modificar y trabajar con materiales, *shaders*, sonidos, animaciones y otras muchas herramientas, cada tipo de *asset* tiene su propia pestaña con las características intrínsecas del propio objeto.

Por el contrario Unity, tiene una interfaz un poco más intuitiva en comparación. La mayoría de las herramientas de visualización y modificación de *assets* no vienen instaladas de forma predeterminada por lo que es visualmente más sencillo e intuitivo.

Respecto a la accesibilidad del propio código, el código fuente de Unreal se encuentra disponible de forma gratuita en GitHub [33], aunque técnicamente el motor no es de código abierto. Esto permite modificar cualquier aspecto del motor, incluso permitiendo modificarlo por completo desde sus bases. Por otro lado, Unity no permite acceso a su código, con la excepción de aquellos que pagan una licencia especifica que permite acceder a este código fuente. Ambos motores se han escrito en C++.

Respecto a la programación de la jugabilidad, Unreal posee dos formas de programación. La programación más tradicional se realiza en C++; esto, unido al permiso implícito de modificación del motor, da una gran libertad, puesto que permite el uso de cualquier librería de C++. La principal desventaja de usar este lenguaje como base son los tiempos de compilación que son comparativamente más largos. Otro de los problemas que nos encontramos a la hora de programar en C++ para Unreal es la documentación, no tiene demasiados ejemplos y a veces puede resultar un poco complicado encontrar la información que necesitas. Este problema se suple con la documentación que hay en torno a la segunda forma de programar en Unreal: los *Blueprints*.

Los *Blueprints* son un lenguaje de *scripting* visual. Este tipo de programación por nodos es una de las características estrella de este motor, ya que permite una mayor accesibilidad para aquellas personas que no tienen un perfil tan técnico. Más adelante hablaremos de ello y las ventajas y desventajas que supone su uso respecto a la programación más tradicional.

Respecto a la programación en Unity, el lenguaje principal de la lógica del juego es en C#, originalmente soportaba otros lenguajes como Boo o Python pero han perdido su uso y se están centrado en el desarrollo con C#. Actualmente están trabajando en una implementación del lenguaje mono y están migrando hacia una implementación DOTS (*data oriented technology Stack*). Los tiempos de compilación de Unity son significativamente más rápidos. La principal desventaja es que aunque posee lenguaje de *scriptado*, este ha sido desarrollado por la comunidad de Unity y no esta implementado de forma nativa en el motor.

2.5.4 Herramientas

Ambos motores proporcionan diferentes recursos. Unreal proporciona de forma nativa todas las herramientas que puedas necesitar: Editores de terreno, herramientas para la optimización de los LODs [2], entre muchas otras herramientas [30] En comparación, Unity no tiene tantos recursos y las que hay implementadas en el motor pueden dar problemas. Las actualizaciones que recibe Unity con más escasas y peor implementadas que en Unreal.

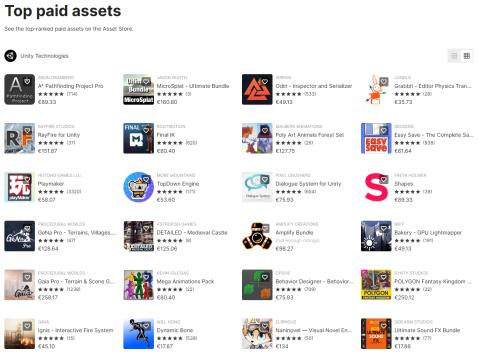
El principal inconveniente de trabajar con Unreal, es que estas herramientas y el formato de la herencia de clases que tiene esta pensado para un formato especifico de juegos, que son los *shooters* (juegos de tiros) en primera y tercera persona. Si tu juego requiere unas mecánicas que se salgan un poco de lo convencional te puede dar un poco más de trabajo implementarlo. Como Unity no fuerza esa estructura de clases, te permite diseñar las herencias en función del proyecto que vayas a crear [26].

Esto se ve claramente en la creación de juegos 2D. Originalmente Unreal no se diseño para este formato de juegos y aunque tiene herramientas disponibles no están tan pulidas como las herramientas para el 3D. Por lo que si tu juego va a realizarse en 2D la mejor opción sin duda es Unity.

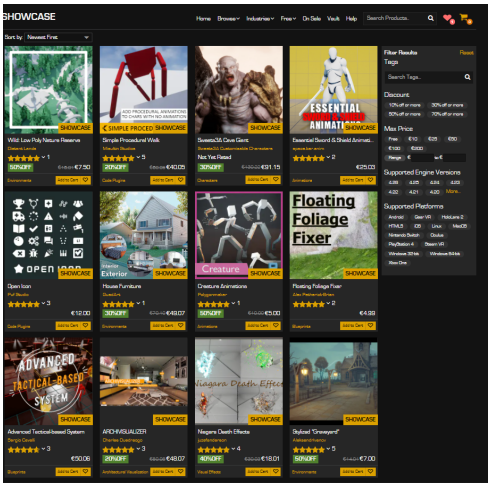
[2] los LOD, Level Of Detail por sus siglas en inglés, son un sistema de optimización que ajusta el nivel de detalle de un modelo 3D en función de la distancia de renderizado de tal forma que, cuanto más lejos este el modelo de la cámara de renderizado, menos detalles tendrá y menos trabajo realiza el motor para renderizar el modelo en cuestión.

2.5.5 Comunidad y Marketplace

Otra de las cosas a tener en cuenta son las comunidades de cada motor. Ambos motores tienen una gran comunidad de desarrolladores con cursos tanto gratuitos como de pago. Sí que es importante destacar que quizás la comunidad de Unity es ligeramente superior y se han creado más libros y tutoriales,lo que es una consecuencia lógica teniendo el cuenta que Unity se desarrolló pensando en un mercado más amplio, como es el desarrollo *indie*. Aun así, ambos motores tienen una comunidad lo suficientemente grande como para obtener respuestas técnicas a problemas con más o menos facilidad, ya sea por parte de la comunidad como de la propia distribuidora.



(a) Unity AssetStore



(b) MarketPlace de Unreal

Figura 2.7: Ejemplo de Ambas tiendas

Además de los tutoriales y cursos disponibles, ambos motores poseen una tienda donde creadores de contenido independientes pueden subir diferentes recursos para el motor(ver Figura 2.7). Ya sean *assets* o funcionalidades, como sistemas de diálogo entre otros. El concepto original de tienda de *assets* nació en Unity y es uno de los motivos por el cual, en comparación, la tienda de Unity tiene una mayor cantidad de objetos y recursos. Una indiscutible ventaja de la tienda de Unreal es que cada mes Epic Games pone a disponibilidad del usuario varios *assets* de la tienda de forma gratuita(ver Figura 2.8) y para siempre, por lo que, con el paso del tiempo, un desarrollador acaba teniendo una biblioteca de *assets* de alta calidad para sus juegos.

2.5.6 Políticas de uso y pago de licencias

Por último, vamos a hablar de las políticas de uso y los pagos de licencia. Ambos motores tienen unas políticas muy similares. Inicialmente ambas licencias son gratuitas y, si el producto no obtiene ningún beneficio económico, no se tiene que pagar ningún tipo de compensación económica.

Para el caso de Unity, existen cuatro tipo de licencias [34], elegibles en función de los beneficios obtenidos por el producto en los últimos doce meses. Los precios varían desde el plan personal (completamente gratis)hasta los dos mil dólares al mes para diez licencias. En el caso de las opciones de pago, la única opción que permite el acceso al código fuente del motor sin ningún pago adicional es la opción profesional. Todas las características y beneficios de cada licencia se pueden ver en la Figura 2.9.

En el caso de Unreal ofrecen diferentes licencias en función del uso que se le va a dar al motor. Distinguen dos usos principales: el desarrollo de juegos y la creación de contenidos. La licencia es comple-

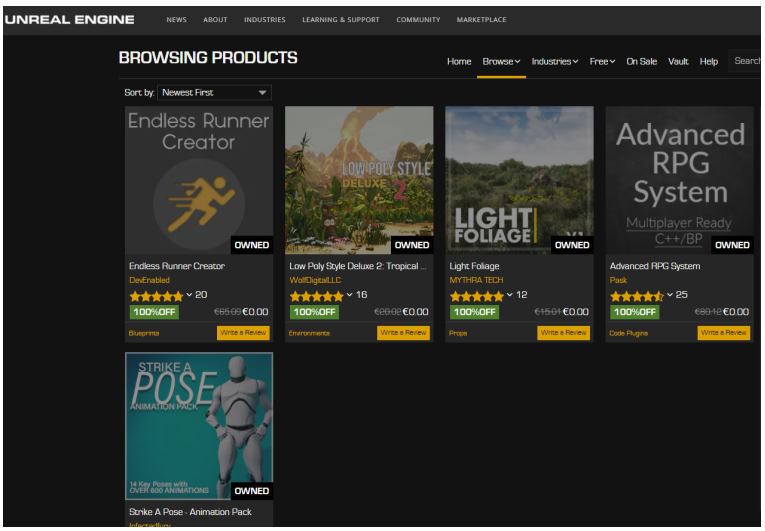


Figura 2.8: *Free Of The Month* en el Unreal Engine *MarketPlace*

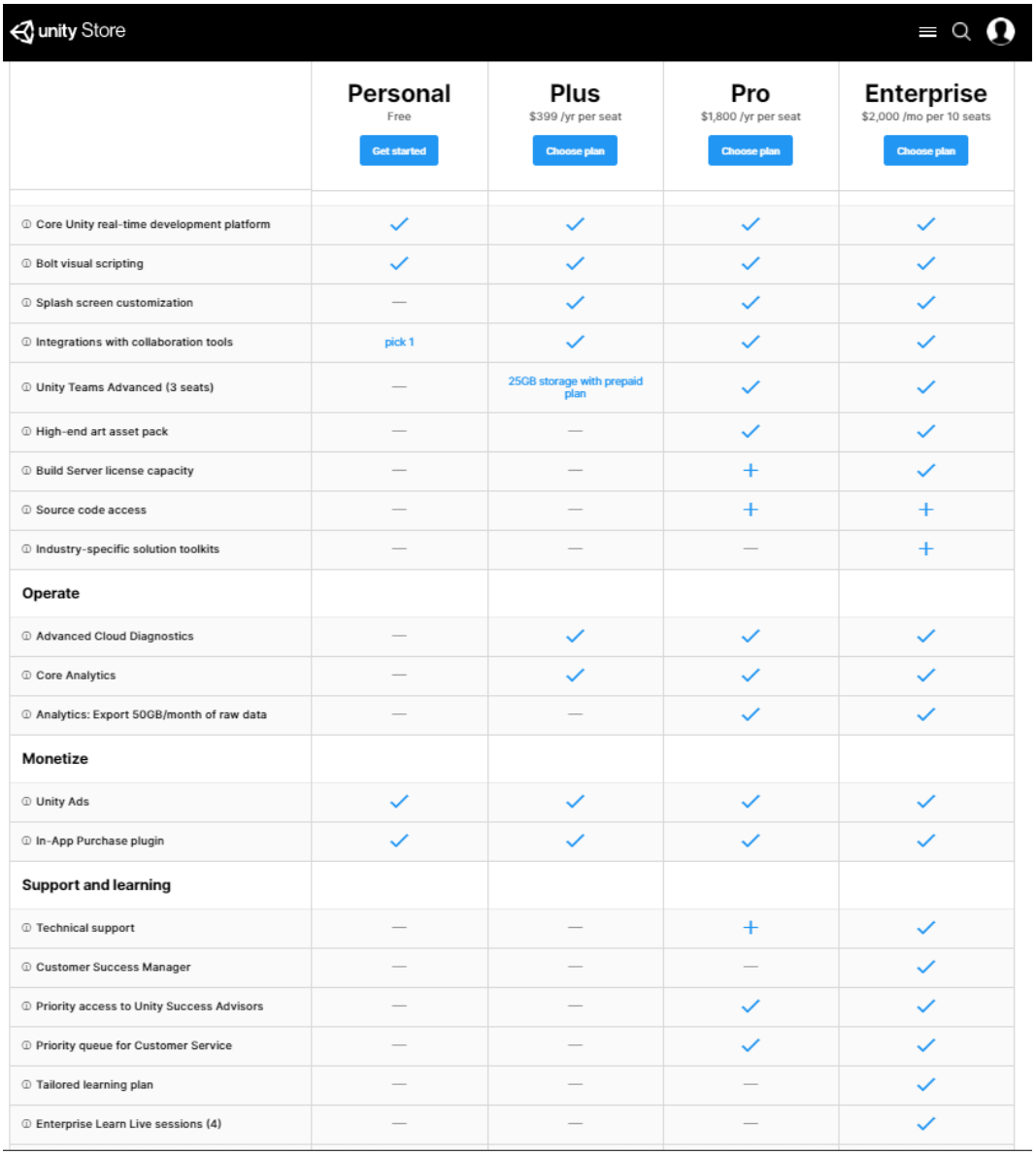
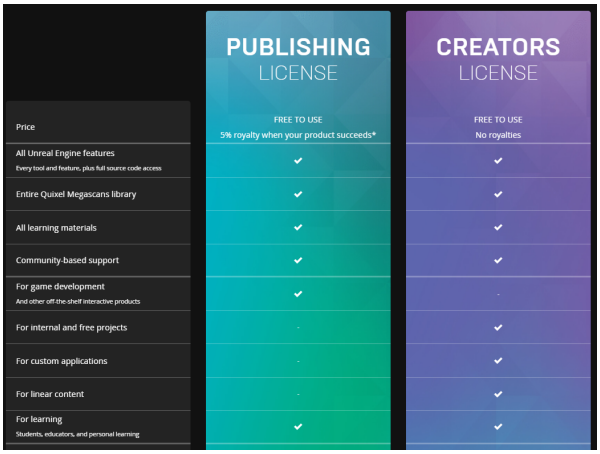


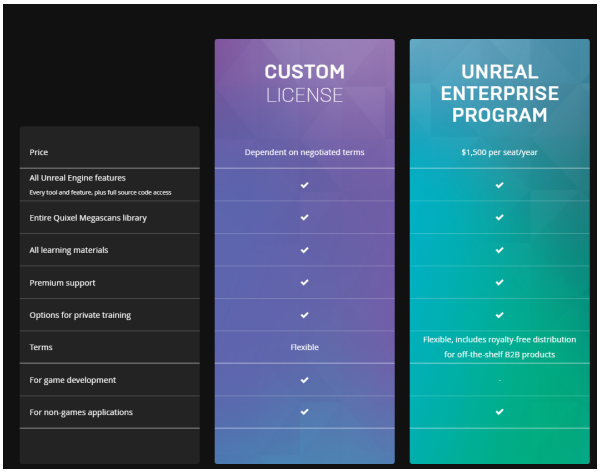
Figura 2.9: Licencias de Unity

tamente gratuita para la creación de contenidos (tutoriales, libros, *assets* para el *marketplace*, creación de proyectos internos), el requisito indispensable para usar esta licencia es que no se puede publicar [35]. Para el caso de la creación y desarrollo de juegos cuyo objetivo sea obtención de beneficio económico [36], el pago de la licencia es del cinco por ciento de los beneficios una vez el producto alcance el millón de dólares. Los requisitos y ventajas para estas dos licencias se pueden ver en la Figura 2.10a.

Además de estas licencias, Unreal ofrece otras dos opciones para aquellos casos en los cuales no se usa el motor de la forma mas *tradicional* [37]. Como se ha comentado anteriormente, el código fuente del motor esta disponible para leer y modificar. Para aquellos casos en los que se desea modificar el motor parcialmente usando parte del código fuente del mismo, Epic Games ofrece una licencia personalizable que se negocia con ellos hasta llegar a un acuerdo. El último caso es cuando se quiere usar el motor pero para el desarrollo de juegos, es decir, cuando se usa el motor para cuestiones de diseño arquitectónico, ejemplos de simulación de físicas y de partículas entre muchas otras. Para estos casos Unreal ofrece una licencia especial denominada *Unreal Enterprise program*, con un coste de mil quinientos dolares al año por licencia. Las ventajas y usos de estas dos licencias se pueden ver en la Figura 2.10b



(a) Licencias Juegos Unreal



(b) Otras licencias adicionales

Figura 2.10: Diferentes licencias del Unreal

2.6 Elección final

Este proyecto va a tener un uso académico y no se tiene intención de obtener beneficio económico, por lo que cualquiera de los dos motores serían igual de válidos. Algo también a tener en cuenta es la limitación temporal y la fuerza de trabajo. Bajo estas condiciones Unreal parece ser la mejor opción, puesto que al proporcionar todos los módulos ya instalados permite una configuración y puesta en marcha mucho más rápida que Unity. Hay que comentar que, a pesar de las ventajas y desventajas de cada motor, ambos desempeñan de forma notable y, por tanto, una razón a tener en cuenta a la hora de tomar la decisión final puede ser una cuestión de preferencias.

Capítulo 3

Unreal Engine 4

Durante la siguiente sección se va ha explicar en detalle las características del motor gráfico Unreal. Además, se va ha desarrollar los pasos a seguir para poder instalar y usar el motor. Para ello es necesario aclarar los requisitos previos a la instalación.

3.1 Instalación del motor

3.1.1 Prerrequisitos

El requisito [38] indispensable a la hora de poder utilizar Unreal como herramienta de desarrollo es la instalación de un compilador de C++. Este compilador es el visual studio 2019 para Microsoft, para el sistema operativo de Windows y itunes19 si se trabaja con MacOS (ver Figura 3.1).

Unreal genera de forma automática los archivos .Build.cs y .Target.cs cada vez que se genera un nuevo proyecto. Estos archivos se pueden modificar y personalizar para crear diferentes configuraciones. En las figuras 3.2 y 3.3 se pueden ver las configuraciones de ambos archivos de este trabajo.

Además de la instalación del compilador, los requisitos hardware recomendados [38] son un procesador Quad-core intel o AMD, con una velocidad mínima de 2,5 GHz, una memoria RAM mínima de 8GB y una versión de Directx11 o superior (ver Figura 3.4).

3.1.2 Instalación de Unreal

Para la instalación del motor, hay una guía paso a paso en la página oficial de Unreal [39]. Inicialmente, es necesario descargarse e instalar el *launcher* (ver Figura 3.5). Para poder descargarlo es necesario seleccionar el tipo de licencia que se va a usar [40], en nuestro caso la licencia de publicación [35]. Tal y como se ha explicado anteriormente, esta licencia es completamente gratuita, siempre y cuando los beneficios anuales sean menores a un millón de dolares. No se puede seleccionar la licencia de creadores ya que, a pesar de que el carácter del proyecto esta orientado a la educación, esta formación no está relacionada con el aprendizaje y uso del motor.

Después de seleccionar el tipo de licencia, hay que elegir la carpeta en la que se va a guardar el *launcher*. Una vez se ha completado la descarga podemos pasar a ejecutar el instalador.

Para poder instalarlo hay que ir siguiendo los pasos que aparecen en pantalla. Una vez ha terminado la instalación es necesario iniciar sesión con una cuenta de Epic Games. En el caso de no tener ninguna

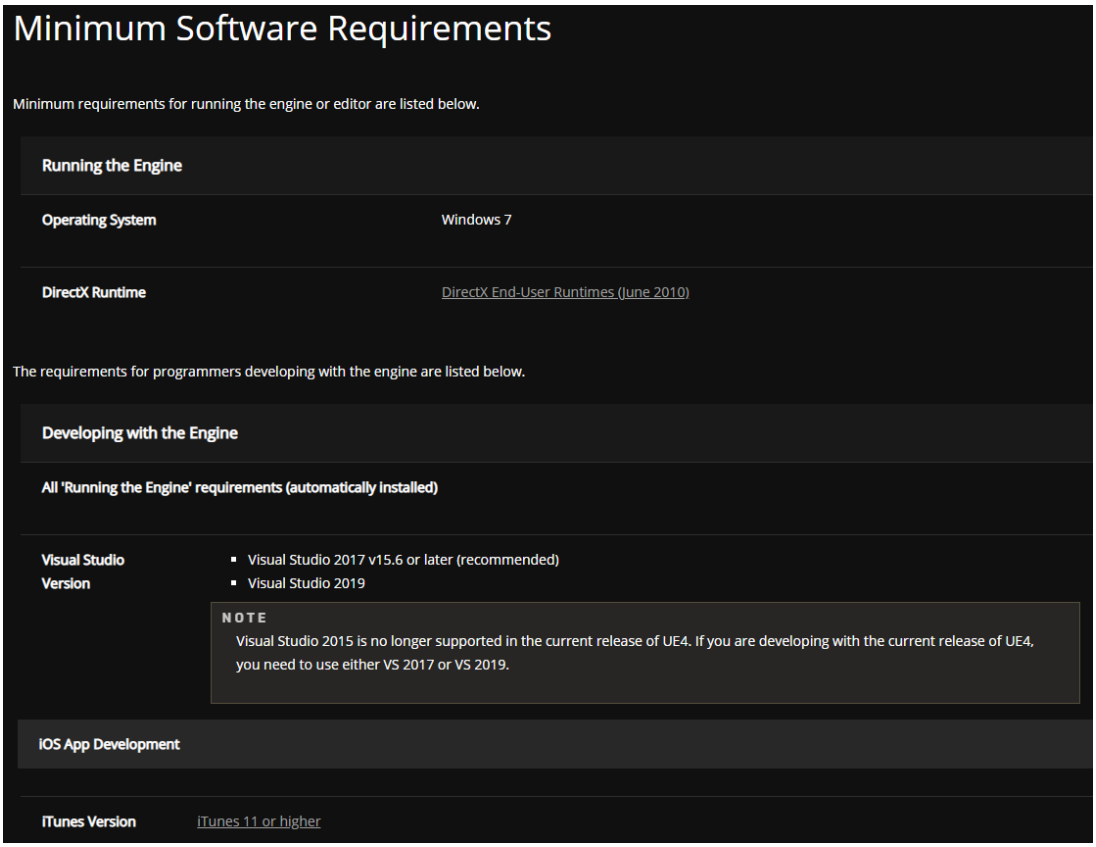


Figura 3.1: Requisitos minimos software

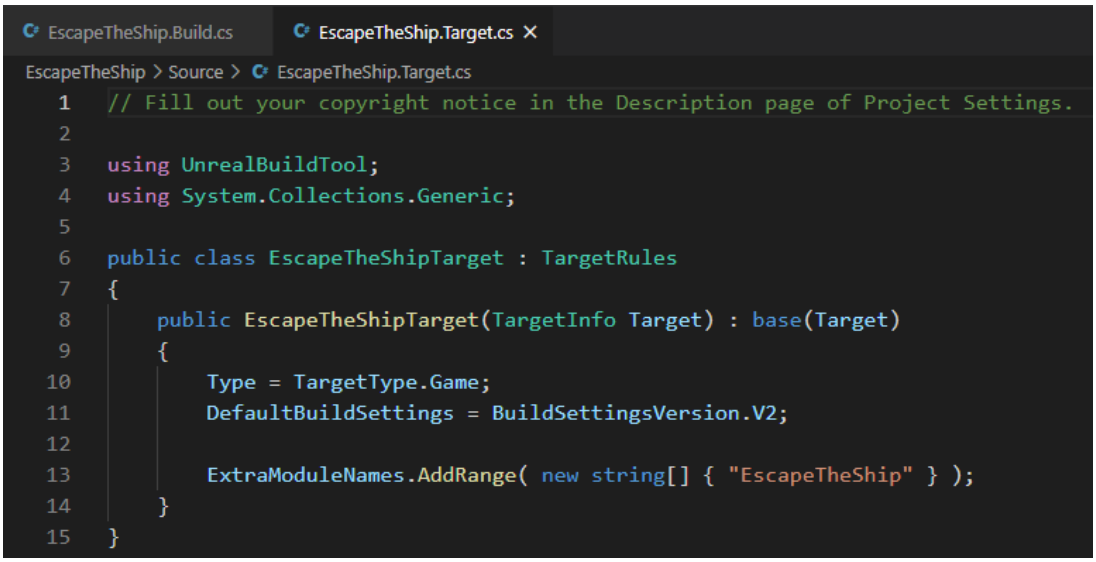


Figura 3.2: archivo target.cs

EscapeTheShip.Build.cs EscapeTheShip.Target.cs

EscapeTheShip > Source > EscapeTheShip > EscapeTheShip.Build.cs

```
1  // Fill out your copyright notice in the Description page of Project Settings.
2
3  using UnrealBuildTool;
4
5  public class EscapeTheShip : ModuleRules
6  {
7      public EscapeTheShip(ReadOnlyTargetRules Target) : base(Target)
8      {
9          PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
10
11         PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore" });
12
13         PrivateDependencyModuleNames.AddRange(new string[] {  });
14
15         // Uncomment if you are using Slate UI
16         // PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });
17
18         // Uncomment if you are using online features
19         // PrivateDependencyModuleNames.Add("OnlineSubsystem");
20
21         // To include OnlineSubsystemSteam, add it to the plugins section in your uproject file with the Enabled attribute set to true
22     }
23 }
```

Figura 3.3: archivo build.cs

Recommended Hardware

Operating System	Windows 10 64-bit
Processor	Quad-core Intel or AMD, 2.5 GHz or faster
Memory	8 GB RAM
Video Card/DirectX Version	DirectX 11 or DirectX 12 compatible graphics card

Figura 3.4: requisitos hardware recomendados

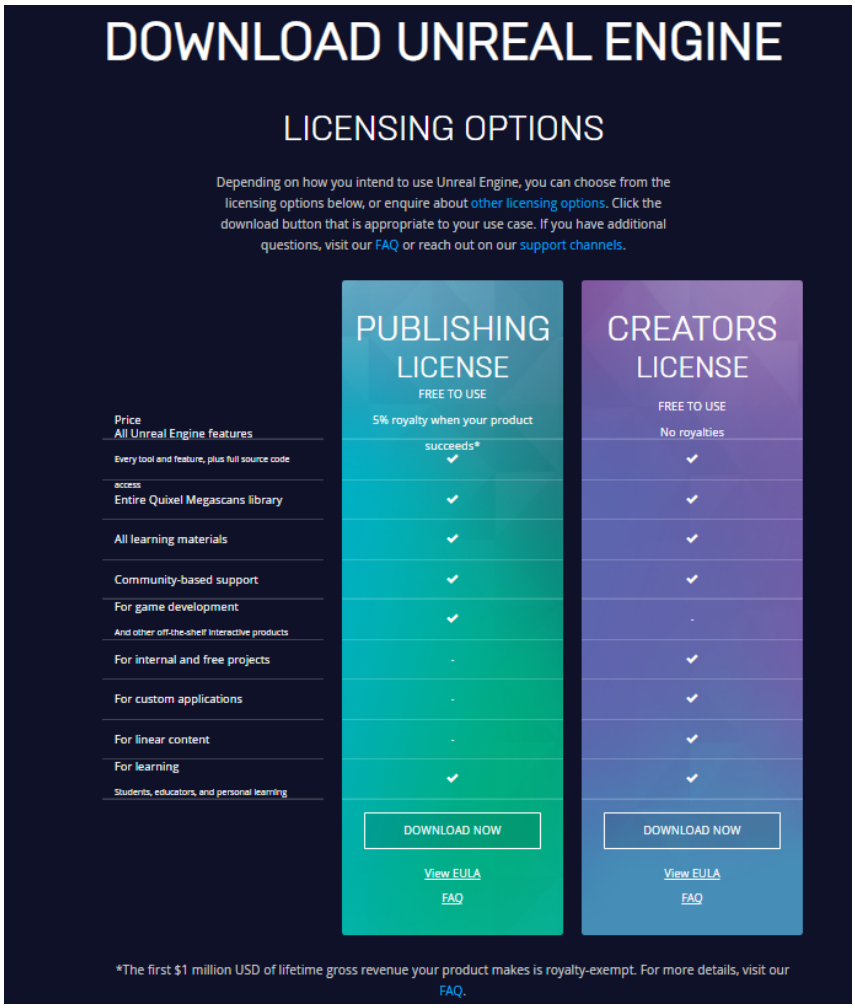


Figura 3.5: Pagina principal launcher de Epic Games

cuenta, hay que crear una. En la Figura 3.6 se pueden ver los diferentes registros para iniciar sesión o crear una cuenta de Unreal.

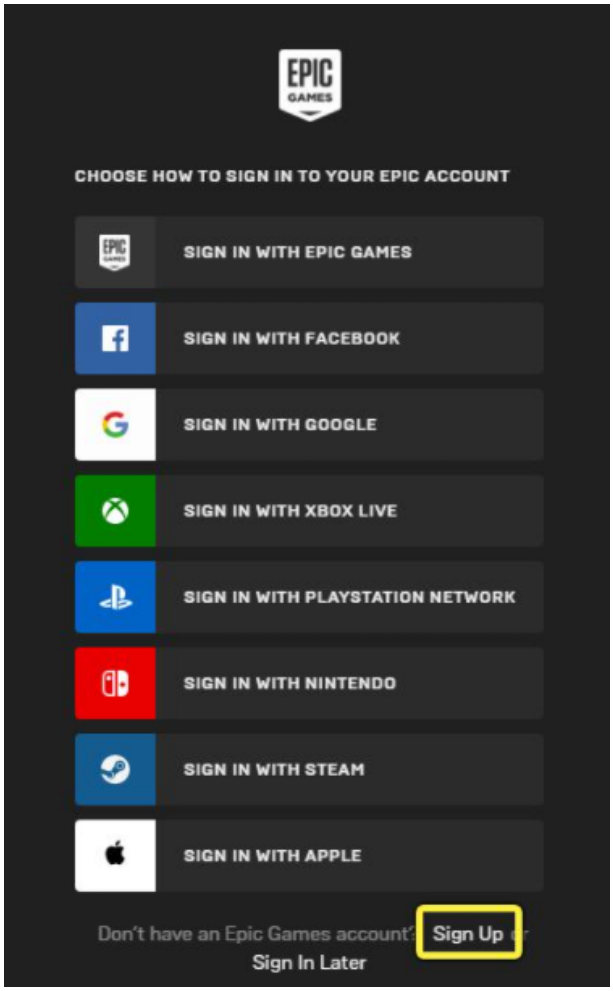


Figura 3.6: Registro y cuenta de Unreal

Una vez nos hemos registrado y autenticado en el launcher, accedemos a toda la información y el registro de nuestra cuenta. En la barra lateral izquierda tenemos una serie de pestañas que nos permiten acceder a diferentes sitios dentro del *launcher*. La primera opción es la página principal, en ella se muestra información y noticias que nos puedan resultar interesantes. A continuación, se puede acceder a la tienda en la que se encuentra todo el catálogo de juegos disponible de Epic Games. En la pestaña siguiente tenemos acceso a la biblioteca, en ella podemos acceder a nuestra lista de productos a los que tenemos acceso.

Por último accedemos a la ultima pestaña, Unreal. En esta pagina encontramos todo lo relacionado con el motor, ultimas noticias, tienda de productos de *assets* (bazar), tenemos otra biblioteca, exclusiva para los proyectos y *assets* del motor, entre otras cosas. Todo esto se puede ver en la Figura 3.7.

3.1.3 Instalación desde el código fuente (Linux)

Para el sistema operativo de Linux hay que construir el motor gráfico desde cero [41]. El motivo es porque la plataforma de Epic Games no tiene ninguna versión precompilada del motor para este sistema operativo, por lo que hay que descargar el código fuente desde GitHub. Al igual que ocurre con las

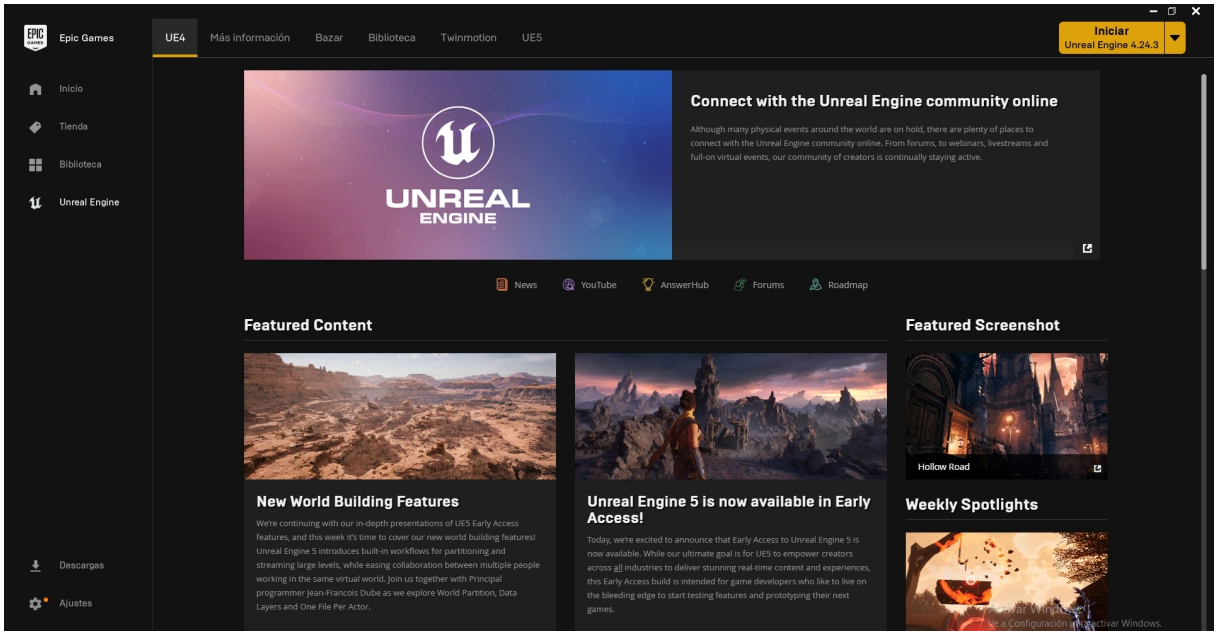


Figura 3.7: Pagina principal launcher de Epic Games

versiones de Windows e IOS, es necesario crear una cuenta valida de Epic Games. Los pasos a seguir para crear una cuenta son los mismos que en Windows y IOS.

Para poder descargar el código fuente, además de la cuenta de Epic Games, debemos tener una cuenta de GitHub. Una vez creadas la cuenta de GitHub y Epic Games, vamos a nuestra cuenta de Epic Games y dentro de ajustes le damos a conectar Cuentas (ver Figura 3.8), seleccionamos el icono de GitHub y autorizamos la conexión en la cuenta de GitHub. Como último paso, previo a la descarga, hay que configurar Git en el dispositivo en el cual vayamos a trabajar con el motor. Como se puede ver, además de conectar la cuenta de GitHub tambien podemos conectar la cuenta con Nintendo, xbox y Playstation, estas conexiones son necesarias si se quiere desarrollar algún producto para estas plataformas.

Una vez conectamos las cuentas, desde la pagina de GitHub nos llega una invitación por parte de Epic Games, para poder acceder a todo el contenido de su página en GitHub (ver Figura 3.9a). Una vez aceptamos la invitación podemos acceder a todo el contenido de su pagina en GitHub, lo cual incluye el repositorio privado de Unreal, tal y como se puede ver en la Figura 3.9b.

Una vez se ha configurado todo, se puede descargar el código fuente, desde la pagina de GitHub de Unreal. Hay dos formas de obtener el código, se puede descargar el código o clonarlo en un repositorio local [42]. Para descargarlo, desde la propia pagina de GitHub descargamos el código en formato ZIP. Para ello se selecciona la versión del motor que queremos obtener en la pestaña de *release*, tal y como se ve en la Figura 3.10. Una vez elegida la versión, hacemos click en descargar como ZIP. Por último, descomprimimos el archivo ZIP en nuestro disco duro.

La otra forma de obtener el código es clonando el repositorio en nuestro disco duro. Al igual que para la descarga, debemos seleccionar la versión del motor que queramos, después seleccionamos la pestaña de clonar o descargar y copiamos la url. A continuación, usamos el comando git clone con la url que hemos obtenido, dentro de la carpeta en la que deseemos tener el motor.

Si hemos seguido los pasos correctamente, deberemos tener una carpeta del motor en nuestro disco duro. Es necesario tener al menos 100 GB libres en disco para poder realizar la correcta instalación del motor. Si cumplimos los requisitos, dentro de la carpeta raíz del repositorio ejecutamos el comando *./setup.sh* esto genera los archivos necesarios para generar los archivos del proyecto, a continuación y en

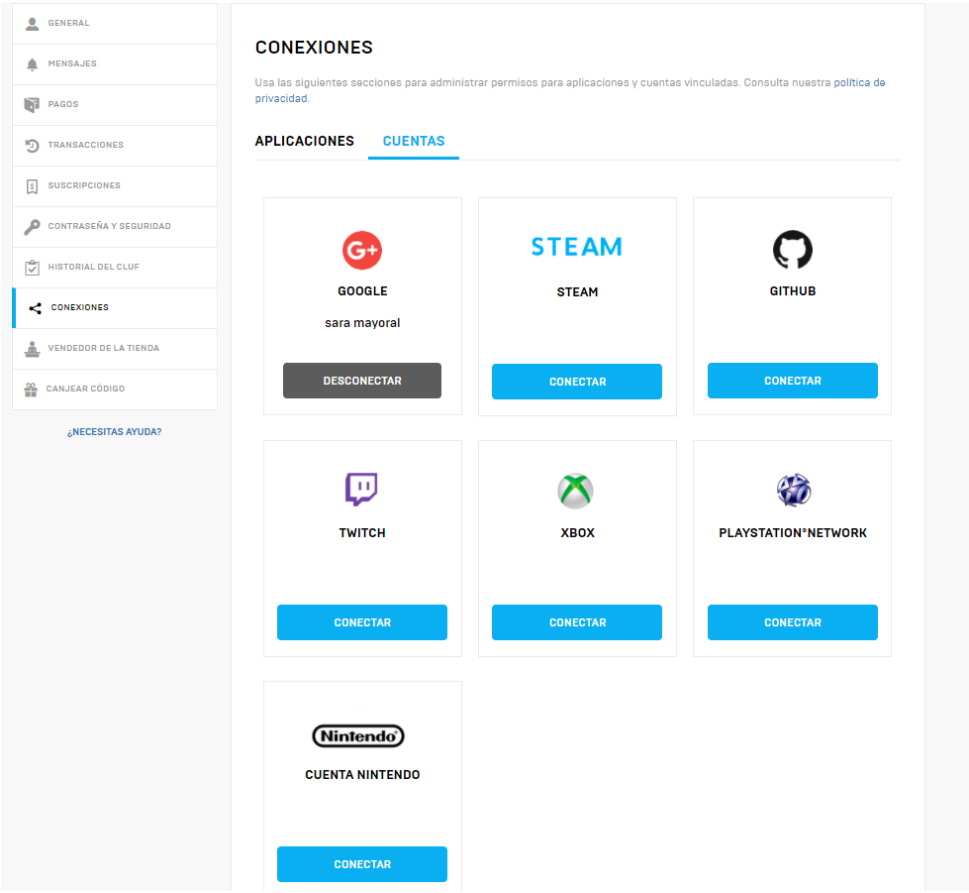
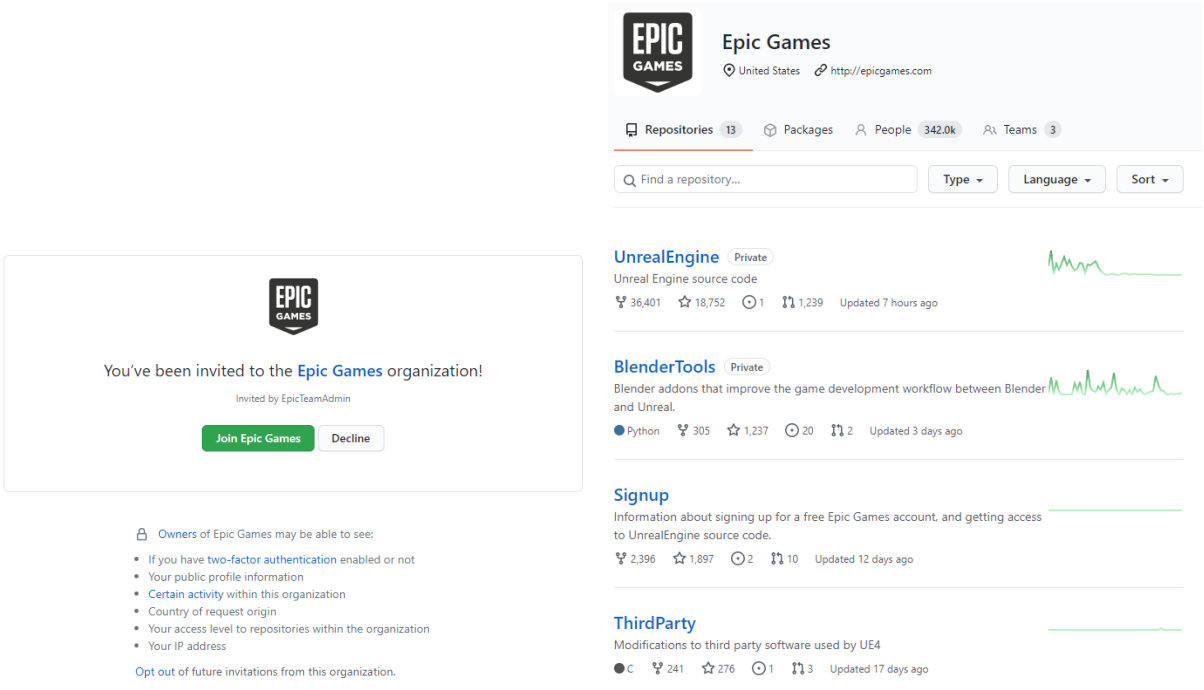


Figura 3.8: Pagina de epic game para conectar diferentes cuentas



(a) Invitación Epic Games en GitHub (b) Repositorio Epic Games

Figura 3.9: Invitación y repositorio de Epic Games en GitHub

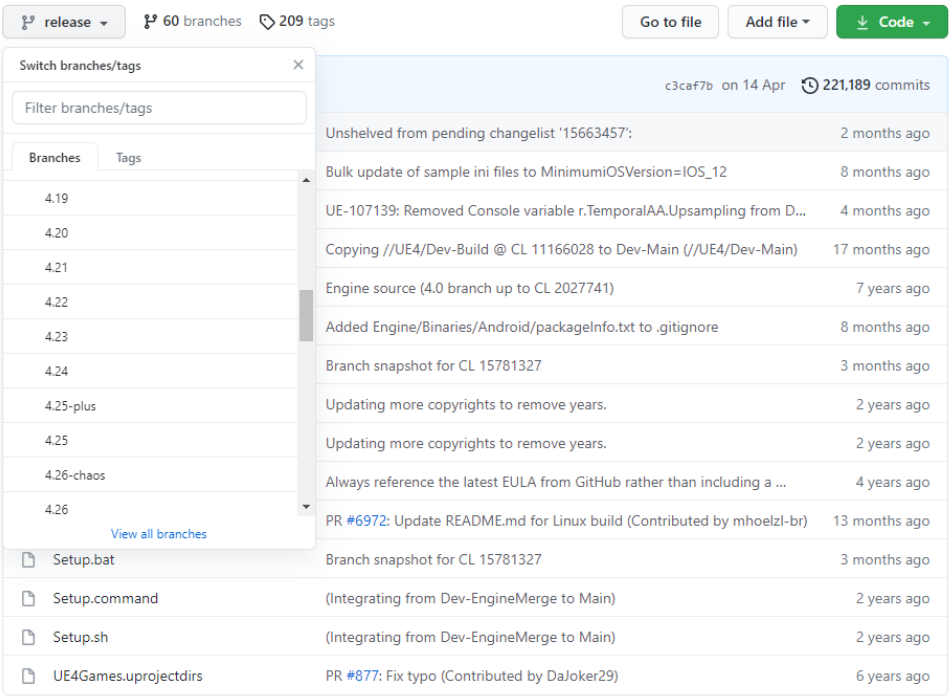


Figura 3.10: repositorio Motor gráfico Unreal

la misma carpeta ejecutamos el comando *./generateProjectFiles.sh*; por último, ejecutamos el comando *make* esto construye el proyecto. Hay que tener en cuenta que en función de las especificaciones técnicas de nuestro sistema la compilación puede durar de diez minutos a una hora. Las recomendaciones de Epic Games es de tener mínimo 8 GB de ram y un procesador de al menos ocho núcleos (incluyendo *hyperthreading*) [41].

Una vez se ha terminado de construir todo pasamos a la ejecución del motor. Para poder hacerlo debemos situarnos en la carpeta *Engine/Binaries/Linux/*. Una vez en ella ejecutamos el comando *./UE4Editor*. Estos pasos hay que realizarlos cada vez que se quiera acceder al motor. Hay que tener en cuenta que la primera vez que ejecutemos este comando el motor tardará un poco mas de lo habitual ya que necesitara compilar los *shaders* por primera vez.

3.1.4 Control de versiones

Una consideración a tener en cuenta, sobre todo para proyectos grandes en los que participa mucha gente, es el control de versiones [43]. De forma nativa Unreal nos permite gestionar las versiones de nuestro proyecto con Git. Para poder configurarlo, desde las preferencias de nuestro proyecto, nos dirigimos a la pantalla de Carga y guardado, en ella encontramos la pestaña source Control, allí podemos configurar y activar el control de versiones(ver Figura 3.11). Otra forma de activar la versión de controles es desde el buscador de contenido. Para ello hay que hacer *click* el botón izquierdo del ratón dentro del menú contextual y pulsar en conectar a control de versiones, tal y como se muestra en la Figura 3.12. Al hacerlo se abrirá un menú contextual en el cual se pueden configurar el *host*, usuario y el proveedor del control de versiones, en el se incluye también una forma de autenticarse en caso de ser necesario.

Una vez se ha activado el control de versiones, todos los archivos tendrán un identificativo en la esquina superior derecha que especifica su estado. Un tick rojo si el archivo esta actualizado por el usuario un tick azul si esta actualizado por otro usuario. Una cruz roja para aquellos archivos que están marcados

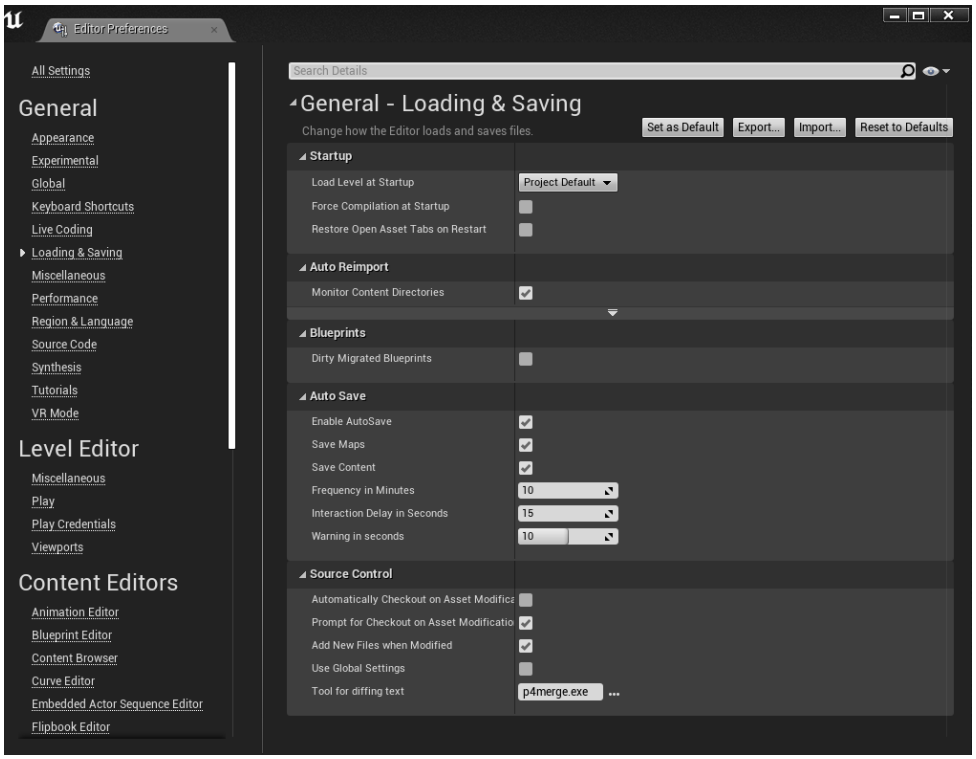


Figura 3.11: Pantalla de carga y guardado

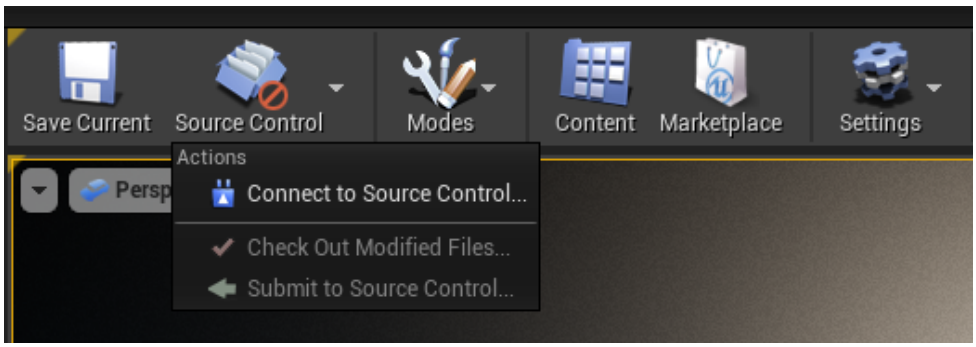


Figura 3.12: Menu contextual para activar control de versiones

para añadir, una interrogación amarilla indica que el archivo se ha modificado y no se ha añadido para la subida y por ultimo una exclamación amarilla indica que existe una versión nueva en el repositorio.

Otra de las características de activar el control de versiones es que, si hacemos *click* con el botón derecho en cualquiera de los *assets*, se abre un menú contextual en el que están disponibles las opciones de verificar, actualizar, el historial y diferencias entre la versión en local frente a la existente en el repositorio.

3.2 Interfaz y *framework*

Una vez se han realizado los pasos para la instalación, pasamos a la explicación de la interfaz y el área de trabajo. También se van a explicar ciertos conceptos y terminologías básicas del motor que permiten una mejor comprensión de su funcionamiento.

3.2.1 Creación de un nuevo proyecto

Para empezar a trabajar en Unreal se debe crear un proyecto [44]. Este proyecto es un conjunto de carpetas que contiene todo el código e información que se va creando dentro del editor, independientemente del tipo de proyecto que sea. Además de las carpetas el proyecto viene referenciado por un archivo con la extensión *.uproject*, este archivo es dependiente del directorio de carpetas, su principal uso es la referencia a la creación, guardado y carga de objetos dentro del proyecto.

No hay número límite a la hora de crear diferentes proyectos, todos ellos se pueden modificar y mantener en paralelo, cada uno independiente del resto ya que tanto el motor gráfico como el editor pueden cambiar fácilmente entre ellos. Esto permite trabajar en varios proyectos a la vez o tener proyectos de pruebas abiertos mientras trabajas en el proyecto principal.

La primera vez que iniciamos el motor, se abrirá una pestaña similar a la de la Figura 3.13. En ella se puede crear un nuevo proyecto o visualizar los proyectos ya creados. En la mitad superior se visualizan los proyectos con una imagen identificativa y el nombre del proyecto. En la mitad inferior se ven las diferentes opciones para crear un nuevo proyecto.

Unreal Engine diferencia cuatro grandes tipos de proyectos: juegos, proyectos relacionados con la televisión, películas y eventos de realidad aumentada, proyectos de arquitectura, ingeniería y construcción y, por último, proyectos relacionados con diseño de productos, manufacturación y automotores.

Al seleccionar una de estas cuatro opciones nos llevara a una nueva pestaña donde podremos seleccionar el tipo de plantilla inicial de nuestro proyecto, en función del tipo de proyecto seleccionado aparecerán diferentes tipos de plantillas. A continuación, debemos seleccionar los ajustes de nuestro proyecto. En estos deberemos seleccionar si queremos crear un proyecto en C++ o usando *Blueprints*, la calidad de los gráficos, la plataforma objetivo en la que nos vamos a centrar y si queremos que nuestro proyecto comience con contenido inicial. Por último, debemos darle un nombre identificativo a nuestro proyecto.

3.2.2 Terminología básica

Antes de poder explicar la interfaz y el flujo de trabajo, es necesario explicar ciertos conceptos y terminologías usadas para Unreal.

- ***Blueprints* [45]:** Los *Blueprints* en Unreal son un sistema de *scripting* visual. Este sistema usa una interfaz basado en nodos que definen los elementos de la jugabilidad y el comportamiento lógico. Se explicará su funcionamiento y características más adelante.

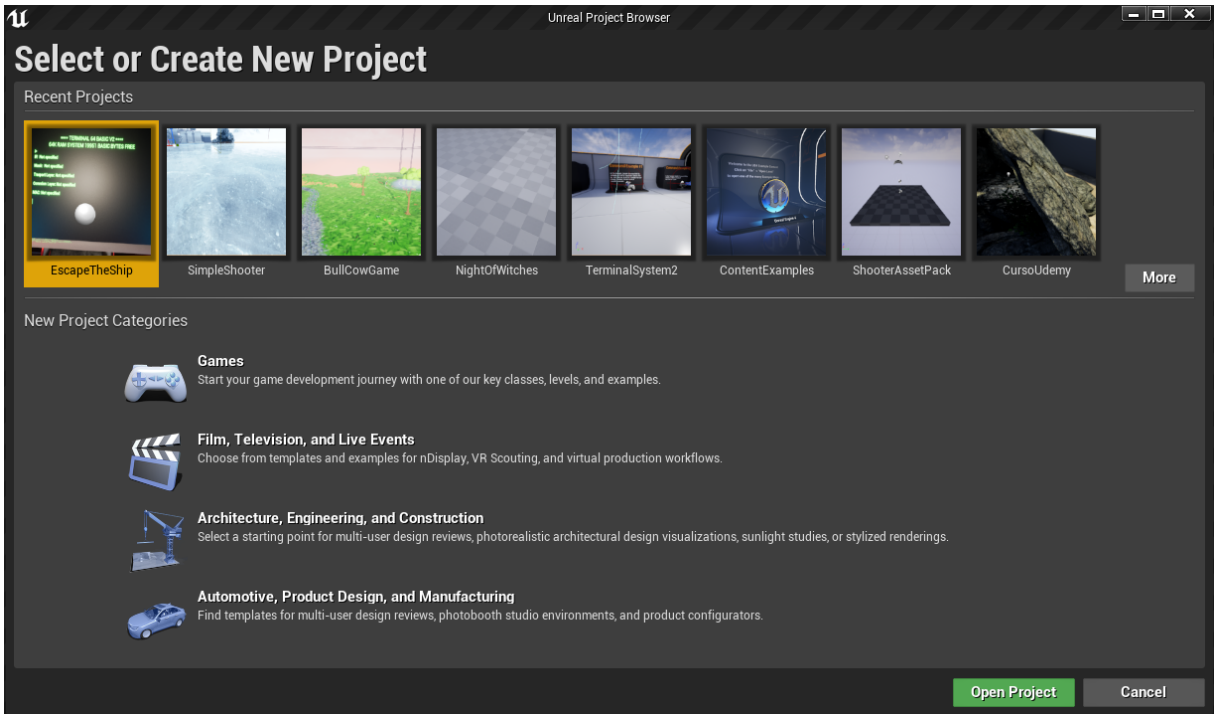


Figura 3.13: Pagina inicial para acceder o crear proyectos

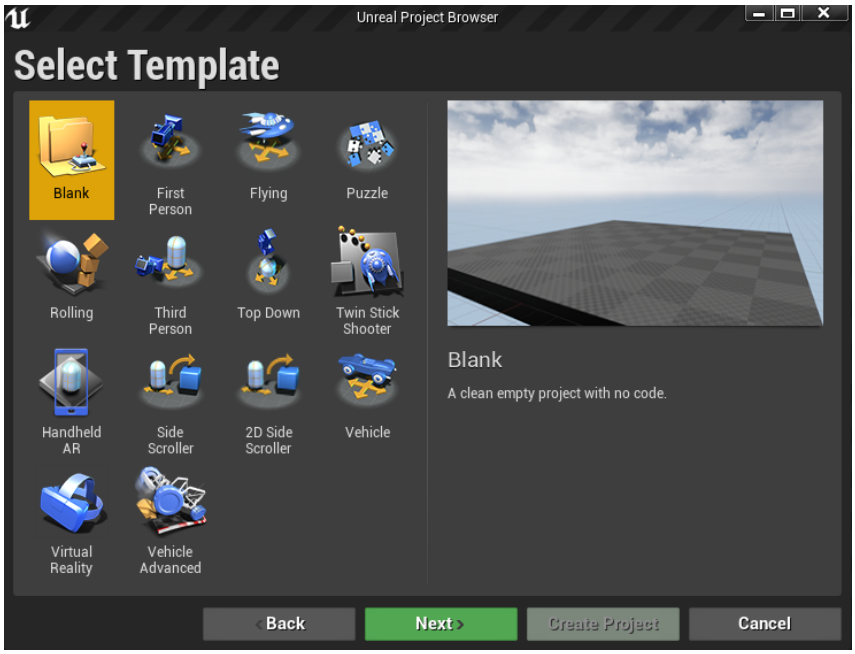


Figura 3.14: Platillas Juegos

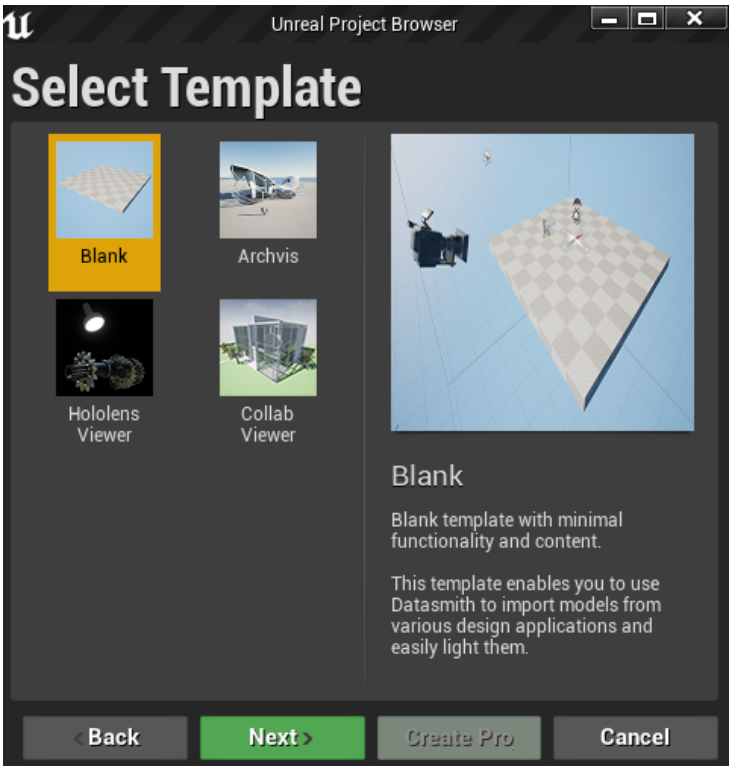


Figura 3.15: Plantillas arquitectura

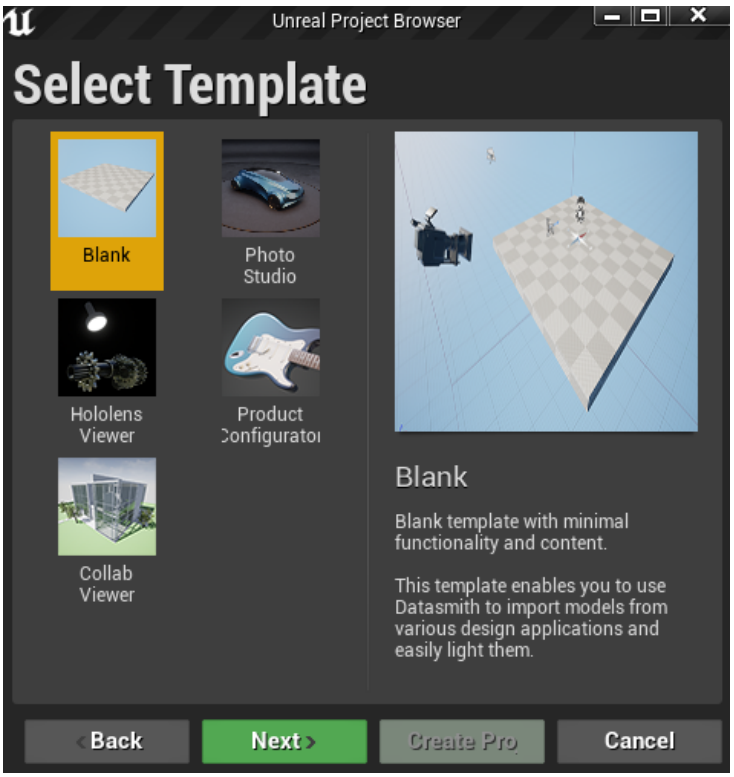


Figura 3.16: Plantillas diseño

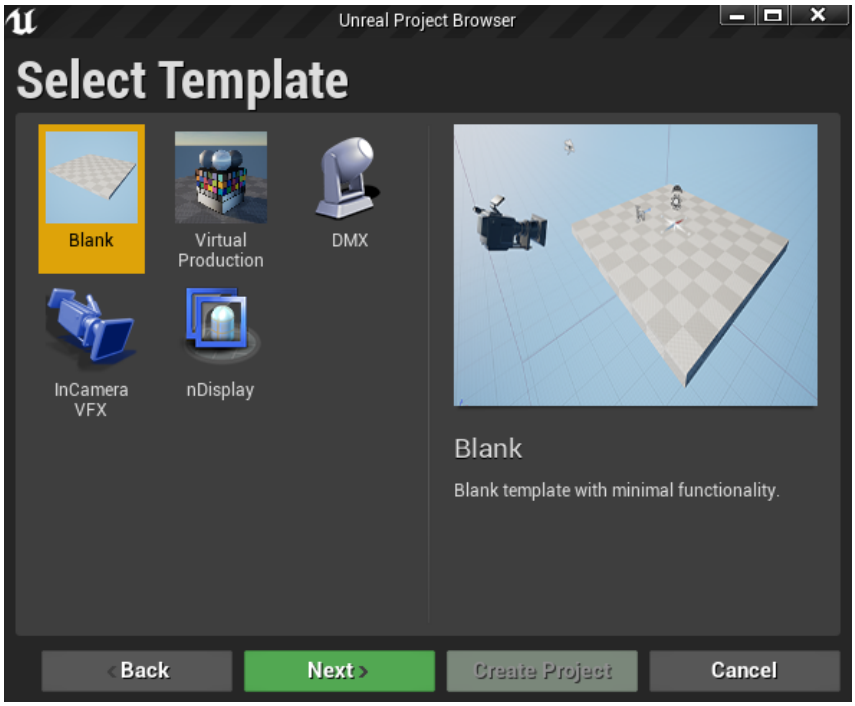


Figura 3.17: Plantillas cine

- **Actor [46]:** Un actor es cualquier objeto que puede ser posicionado en un nivel. Esto incluye cámaras, *assets*, localización de inicio del jugador entre otras cosas. Un actor almacena las transformaciones básica 3D, como la rotación, traslación y escalado. Estos actores pueden ser creados y destruidos a través de la lógica del juego.

- **Componente [47]:** Se trata de una funcionalidad que se añade a un actor. Esta funcionalidad puede ser un componente de luz, un componente de rotación que hace que nuestro actor gire, un componente de audio que añade sonido. Estos componentes no pueden existir en sí mismos y deben están sujetos a un actor para que funcionen.

- ***Pawn* [48]:** Un *pawn* (peón) es una subclase de actor. Este tiene unas características especificas y sirve como avatar dentro del juego, estos peones pueden ser controlados por el jugador o por la inteligencia artificial (IA). Cuando este peón es controlado, ya sea por la IA o por el jugador se considera que esta poseído.

- ***Character* [49]:** En *character* (personaje) es una subclase de peón y esta diseñada para ser usada para el personaje que controla el jugador. Esta clase incluye la configuración inicial de las colisiones, conexión de entrada (*inputs*) para el movimiento bípedo y código adicional para el control del movimiento por parte del jugador.

- ***Player Controller* [50]:** Un *player controller* (controlador del jugador) permite capturar los *input* que emite el jugador y traducirlos en interacciones dentro del juego. Cada juego debe tener al menos un controlador. Además, el controlador es el punto de interacción en los juegos multijugador. En una partida con varios jugadores, el servidor tiene una instancia del controlador por cada jugador, en la parte del cliente tan solo se tiene la instancia correspondiente al jugador en específico. Los controladores de ambos lados son los que se comunican entre sí.

- ***AI Controller* [51]:** El controlador de IA (inteligencia artificial) funciona de forma similar al controlador del jugador. Este controlador posee un peón(*pawn*) o personaje (*character*) no jugable

(este tipo de entidades se denominan NPC- *non playable character*). De forma predeterminada si un peón o personaje no es controlador por el controlador del jugador, el controlador de IA se hará cargo de esa entidad, a no ser que se indique de forma específica.

- ***Player State* [52]:** El estado del jugador, guarda información relacionada con el estado del personaje controlado por el jugador, este estado puede incluir información como el nombre, nivel, vida, puntuación entre muchas otras cosas. Para los juegos multijugador el estado de todos los jugadores existe en todas las máquinas que forman la red, y esta información se puede replicar desde el servidor al cliente para mantener la sincronía de la información.

- ***Game Mode* [53]:** El modo de juego establece las reglas básicas sobre las que se juega. Estas reglas pueden incluir información sobre como se unen los jugadores, si el juego se puede pausar o no, las condiciones de victoria y el comportamiento específico del juego. Este modo de juego puede sobrescribirse para cada nivel, pero solo puede haber un modo de juego por cada nivel. En un juego multijugador, el modo de juego solo existe en el servidor y las reglas se replican en los clientes conectados a este.

- ***Game State* [52]:** El estado del juego es un contenedor que almacena la información que quieres replicar y enviar a los clientes dentro de un juego multijugador. No suele ser habitual usar el estado del juego para juegos de un solo jugador.

- ***Level* [54]:** Un *level* (nivel) es el área donde juega el jugador. Un nivel contiene todo lo que el jugador puede ver y aquello con lo que puede interactuar.

- ***World* [55]:** El mundo (*world*) es el contenedor que almacena los diferentes niveles que se han creado en el juego. Gestiona la carga de estos niveles y los actores y peones dinámicos que se van generando en cada uno de ellos.

3.2.3 Interfaz

Una vez comentados los términos y conceptos básicos de Unreal y habiendo creado nuestro primer proyecto, podemos pasar a la explicación de la interfaz y el editor. Una de las desventajas de este motor es que, para personas nuevas dentro del uso de motores gráficos, su sistema de menús puede llegar a ser complejo. Cada objeto dentro del proyecto tiene su propio menú contextual con características, pestañas y opciones especificas en función del objeto. Esto quiere decir que el menú contextual que se abre para la musica ambiental es diferente al menú contextual de un efecto gráfico. Unreal diferencia diecisiete editores, cada uno con una complejidad variable [56]. En este proyecto se han usado principalmente dos: el editor de niveles y el editor de *Blueprints*, los cuales vamos a explicar a continuación.

Nada más abrir el proyecto se puede ver la interfaz del editor del proyecto. Como se puede ver en la Figura 3.18 se pueden diferencias siete áreas, estas son completamente personalizables, ya que se puede modificar el tamaño, ocultarla a la vista o incluso separarlas en una ventana emergente.

Primero vamos a hablar de la barra de pestañas y el menú, indicados con el número 1 en la Figura 3.18 La barra de pestañas indica con un nombre identificativo qué vamos a encontrar en esa pestaña. Su funcionamiento es similar al de un navegador web, se pueden ir añadiendo diferentes pestañas de los *assets* que se han abierto para modificar. Por otro lado tenemos el menú, al igual que muchos otros programas esta barra da acceso a los menús generales que se suelen utilizar cuando se trabaja en un proyecto [57].

A continuación, tenemos la barra de herramientas [58] (numero 2 en la Figura 3.18), esta barra proporciona un acceso rápido a las herramientas y operaciones que más se suelen usar. Marcado con

Figura 3.18: Interfaz del proyecto

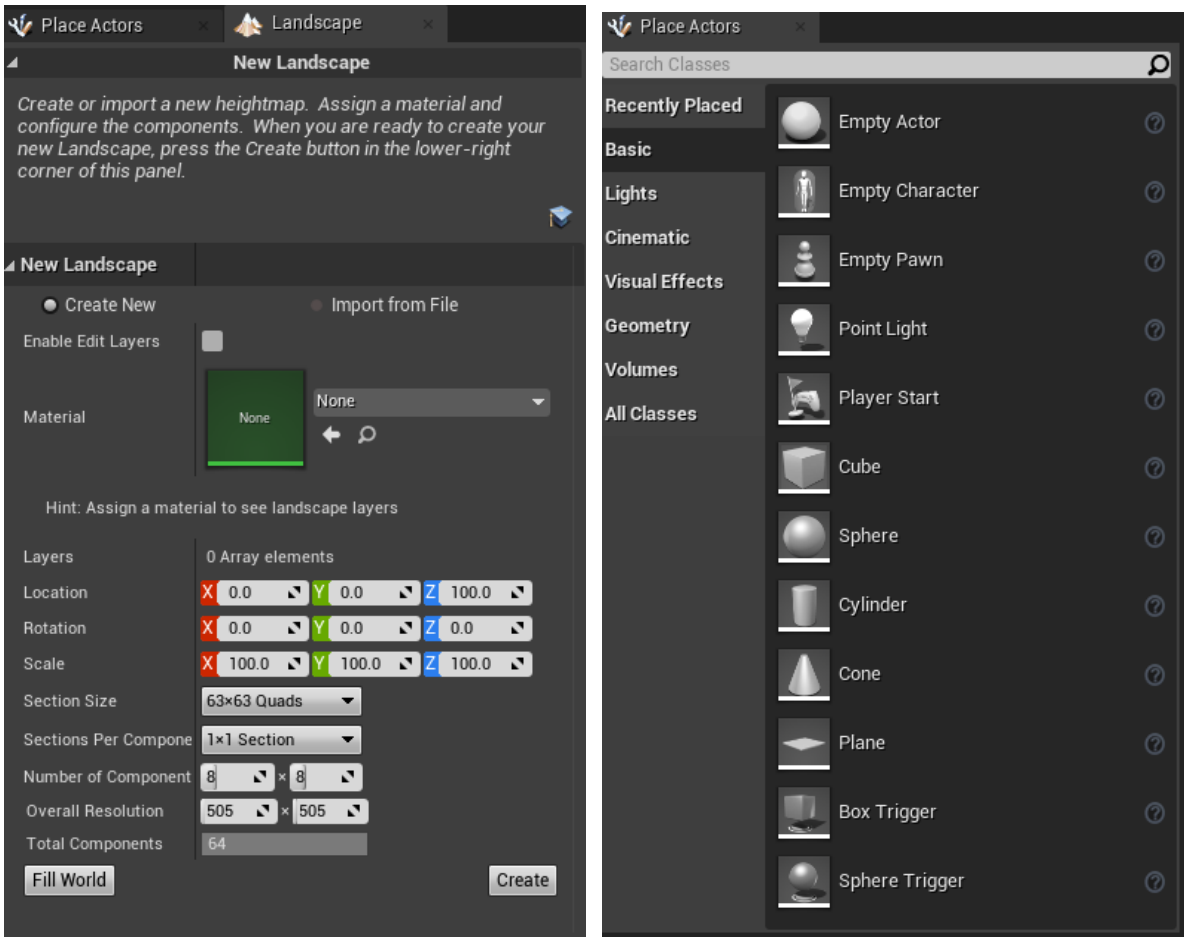
el numero 3 en la Figura 3.18 tenemos una ventana que nos permite añadir actores y objetos dentro del nivel en el que estamos trabajando. Esta pestaña varía en función del modo [59] seleccionado en la barra de herramientas, de esta manera si hemos seleccionado el modo paisaje (*landscape*), esta ventana muestra pinceles que permiten modificar el terreno del nivel (ver Figura 3.19a), si por otro lado queremos posicionar un actor dentro de nuestro nivel, nos mostrará los diferentes actores que podemos añadir (ver Figura 3.19b).

Este menú no es el único que nos permite añadir actores y *assets* a nuestro proyecto, marcado con el numero 5 en la Figura 3.18 tenemos el navegador de contenido [60]. Este navegador es el área principal para importar, organizar, ver y modificar los contenidos del proyecto dentro de Unreal. Permite renombrar mover, copiar y ver referencias.

Justo en el centro de la pantalla tenemos la ventana gráfica [61] (indicada con el número 4 en la Figura 3.18), esta ventana refleja el nivel en el que estamos trabajando. Contiene un conjunto de vistas que reflejan diferentes visualizaciones, en la esquina superior izquierda se puede cambiar el tipo de visualización, indicando, por ejemplo, el tipo de perspectiva. También podemos cambiar el tipo de vista. Por ejemplo, en la Figura 3.20 se puede ver la misma imagen una con la vista de colisiones (figura 3.20b) y la otra con el renderizado y procesado de la iluminación (figura 3.20a).

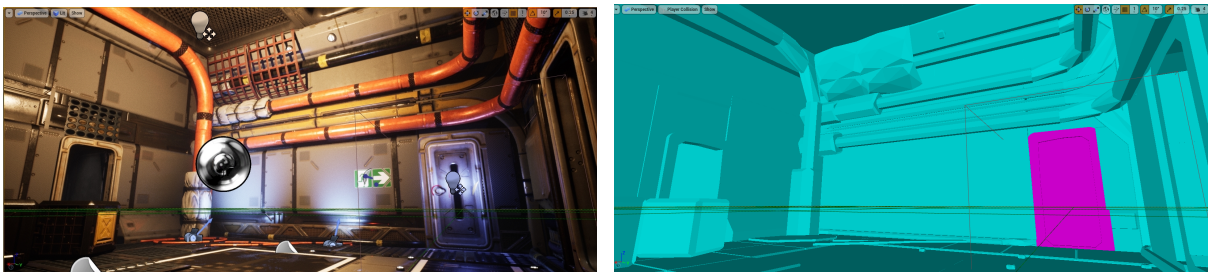
En la parte derecha de la interfaz nos encontramos con una pestaña denominada *world outliner* [62] (numero 6 en la Figura 3.18). Este panel muestra todos los elementos que se encuentran en la escena. Estos elementos se pueden organizar en carpetas personalizables y siguen una estructura jerárquica en forma de árbol. De forma adicional, se pueden crear capas dentro de la estructura de árbol, esto nos permite seleccionar y controlar de forma visual cada capa.

Por último, tenemos la ultima pestaña de la interfaz del editor de niveles. El panel de destalles, denominada con el numero 7 en la Figura 3.18, se abre al seleccionar un *asset* dentro de nuestro nivel. Esta ventana contextual varia su información en función del *asset* ya que indica detalles específicos del mismo. Además, permite hacer pequeñas modificaciones, siempre y cuando la variables a modificar permita ese tipo de acceso. También permite la visualización de las características básicas del *asset*.



(a) Landscape mode (b) Place Actor mode

Figura 3.19: Diferentes Modos en la interfaz



(a) vista iluminación (lit) (b) Vista de colisiones

Figura 3.20: Diferentes vistas

3.3 Lenguaje de programación y *Blueprints*

Una vez explicados las bases y herramientas del motor, pasamos a la programación de la lógica del juego. Como se ha comentado anteriormente, este motor gráfico se programa en C++, otra de las cosas que se han mencionado y que presenta una desventaja comparándolo con Unity es los altos tiempos de compilación. Sin embargo esta no es la única herramienta que posee Unreal para programar. Con el lanzamiento de Unreal 4, se desarrolló una herramienta de *scripting* visual que permite el desarrollo de la lógica de programación, evitando los altos tiempos de compilación.

3.3.1 ¿Qué son los *Blueprint*?

Los *Blueprints* son una herramienta de *scripting* visual basado en nodos. Como la mayoría de los lenguajes de *scripting* se usa principalmente para definir clases u objetos en el paradigma de orientación a objetos. Este sistema es muy flexible a la hora de programar y posee la ventaja de la reducción de los tiempos de compilación, puesto que al ser un *script* la compilación se produce durante el tiempo de ejecución. Otra clara ventaja del uso de esta herramienta es el acercamiento por parte del resto de perfiles que desarrollan un producto de estas características, es decir, no se requiere un perfil tan técnico y resulta más intuitivo que la programación tradicional.

3.3.2 C++ vs *Blueprints*

Una de las primeras preguntas que surgen cuando se comienza a desarrollar un proyecto en Unreal es qué tipo de lógica se va a implementar ¿Se va a usar la programación tradicional o se va a optar por la programación visual usando los *Blueprints*? Esta cuestión puede no ser tan evidente y fácil de responder puesto que cada método posee sus ventajas y desventajas. Sin embargo, se parte de una premisa equivocada, la elección no es **qué** programación elegir, si no **cuándo y en qué momento** [63].

Independientemente de cuál de los dos métodos usemos, estamos escribiendo código, y como tal vamos a tener que enfrentarnos al diseño software que va asociado a cualquier desarrollo. De forma más tradicional, si se intenta llegar a un alto nivel de lógica cómo podría ser el comportamiento de un arma o un misil, es necesario diseñar una lógica a más bajo nivel que va asociado al mismo.

Vamos a usar un ejemplo práctico: queremos diseñar un arma para nuestro juego. Hay muchas cosas a tener en cuenta a la hora de diseñar la lógica. Debemos considerar como acceder a la información dentro de la memoria, cómo va a ser el comportamiento del arma, es decir, la implementación de un comportamiento físico en las balas, qué evento determina cuando se dispara el arma entre otras cosas y, por último, cuál es el aspecto del arma, qué efectos sonoros y visuales van asociados a ella.

Para la lógica de más bajo nivel como el acceso a la memoria, es el propio motor el que se encarga de gestionarlo, aun así, como se ha comentado, el motor permite hacer modificaciones al código fuente por lo que también tendríamos cierto nivel de decisión en caso de ser necesario. Respecto al desarrollo de las clases, en el caso de la programación de las bases del arma, el motor proporciona clases que te dan una base sobre la que trabajar. Teniendo esto en cuenta de formas más o menos intuitiva la estructura de la lógica quedaría de forma similar a la Figura 3.21

Este diseño es una generalización ya que, dependiendo del proyecto y del equipo, esta clasificación puede variar. A grandes rasgos, cuando se habla de programar, por lo general nos referimos a la programación a mas bajo nivel y si hablamos de *scripting* nos referimos a la programación a mas alto nivel. El verdadero potencial del motor reside en la libertad de elección respecto a que se considera alto nivel y bajo nivel. El motor no marca ninguna linea divisoria entre lo que se debe hacer en programación en

Figura 3.21: Grafo de diseño software de un arma

C++ y lo que se debe hacer en *scripting* y tiene herramientas que permiten ambos tipos de programación a casi cualquier nivel.

Volviendo a la discusión original sobre la elección a tomar, si estamos hablando sobre la implementación de herramientas que optimizan el renderizado, la única opción permitida es la programación tradicional, para cualquier desarrollo de lógica que no implique la modificación del comportamiento del motor, cualquiera de las dos opciones es igual de válida, siempre y cuando se tengan en cuenta la ventajas y desventajas de cada una de ellas.

3.3.3 Rendimiento

Para poder explicar las ventajas y desventajas de cada método es necesario explicar unos conceptos básico sobre el funcionamiento de cada uno.

Cuando se programa en C++ el código debe ser compilado antes de poder realizar cualquier prueba. Este código funciona directamente en la CPU. Para el caso de los *Blueprints*, el código funciona a través de un compilador de *scripts*. Este compilador transforma los nodos en una lista unidimensional de código ejecutable. Se trata de un estado intermedio que permite al motor leer el código del *Blueprint* en tiempo de ejecución.

Sabiendo esto, para dos funciones equivalentes, el código en C++ es mucho más rápido de ejecutar ya que esta completamente optimizado a un nivel muy bajo y no hay que realizar traducciones adicionales como ocurre con la ejecución de un *script*. Para solucionar este problema, Unreal ha diseñado una herramienta llamada *"Blueprint nativization"*. Esta herramienta traduce el *Blueprint* a un estado intermedio en C++, que si que permite la compilación directa a código máquina. Esta herramienta no esta pensada para que una vez realizada la conversión se pueda leer o modificar el archivo creado, pero si hay una mejora evidente en cuanto al rendimiento final.

Donde realmente se marca la diferencia el uso de C++ es en los sistemas a bajo nivel, como se ha comentado anteriormente, estos sistemas los decide la propia persona. En situaciones en las que se requiera el procesamiento de muchos datos, como pueden ser el uso de *arrays* grandes o búsquedas, entre otros, la mejor opción suele ser usar C++. También es recomendable usar C++ en clases que tengan muchas instancias o en clases cuyas funciones se ejecuten cada *frame*. Una regla para decidir cual de los dos métodos, es la regla de Pareto [64]. Si una función se va a ejecutar el 80 % de las veces, es recomendable usar C++.

3.3.4 Estructura de clases y dependencias

Otra de las consideraciones a tener en cuenta a la hora de realizar el diseño software es la estructura de las clases y las dependencias entre ellas [63]. A la hora de programar en C++ las dependencias están limitadas en un solo sentido, en el caso de los *Blueprints* no se tiene esa limitación, cada clase *Blueprint* puede referenciar a cualquier otra clase, siempre y cuando esta sea de tipo *Blueprint*.

Como se ha comentado en el apartado anterior, Unreal no hace una distinción clara entre lo que debe ser programado y lo que debe ser *scriptado*. Sin embargo, hay una limitación unidireccional entre dependencias. Una clase *Blueprint* puede depender de una clase de C++, pero no al revés. Los *Blueprints* son *assets*, y el como se organicen entre si es una cuestión más de estructuración de carpetas, debido a la libertad de referencia que tienen los *Blueprints*. En el caso de estar realizando un diseño que modifique el núcleo del juego o estemos realizando una clase programada en C++, sí hay que tener en consideración estas restricciones.

3.3.5 Ventajas y desventajas de ambos

Como se puede ver, ambos sistemas tienen sus ventajas y desventajas. La principal ventaja de los *Blueprints* es la facilidad de uso. Al tratarse de un sistema de *scripting* no hay largos tiempos de compilación lo que permite realizar una mayor cantidad de pruebas. Además, no se tiene la restricción de las dependencias.

En el caso de la programación en C++, las principales ventajas son el rendimiento y el acceso y modificación al núcleo del motor. Además, al tratarse de un lenguaje de programación independiente del motor, permite el uso de librerías externas.

Tal y como hemos mencionado al principio de este apartado, la decisión entre C++ y *Blueprint* no es una o la otra. Unreal permite el uso de *Blueprints* que luego pueden ser desarrollados en C++ una vez se han realizado todas las pruebas y ajustes necesarios. También es perfectamente posible diseñar una clase en C++ de la cual deriven los *Blueprints* que se usarán dentro del juego.

Capítulo 4

Diseño

Para poder poner todos estos conocimientos en práctica se va a desarrollar un proyecto usando el motor gráfico Unreal. Como se ha comentado al principio de este TFG, se va a realizar un caso concreto sobre el potencial de Unreal para la enseñanza, para ello se van a usar los conocimientos adquiridos en la asignatura de Arquitectura de redes (780011) del Grado en Ingeniería de Computadores de la Universidad de Alcalá. El juego parte de la base de que el jugador posee los conocimientos básicos de la estructura de la capa de redes informáticas y por tanto no se realiza ninguna explicación a lo largo del mismo.

4.1 Introducción y objetivos del juego

El jugador se encuentra en una nave espacial, su objetivo principal es arrancar el motor de la nave para poder volver a casa. Para ello deberá recolectar tres baterías que se encuentran dispersas en la nave y que le permitirán acceder a la sala de control. Una vez dentro, accederá a un terminal en el cual deberá realizar la configuración necesaria para poder conectarse a un servidor, del cual sólo sabrá la dirección URL. En la Figura 4.1 se puede ver el diagrama de flujo que sigue el jugador.

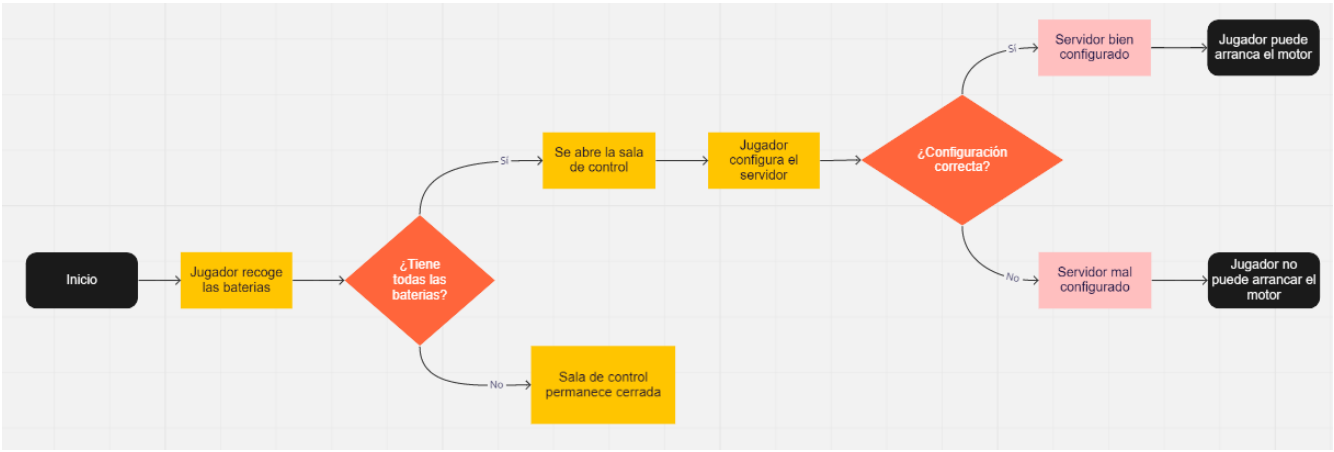


Figura 4.1: Flujo de diseño

4.2 Elementos necesarios

Una vez establecemos las bases y el flujo básico que debe seguir el jugador para terminar el juego, es necesario establecer que elementos hay que crear para poder llevarlo a cabo. Para este proyecto, teniendo en cuenta el flujo de diseño, es necesario diseñar cinco elementos clave: las baterías, la puerta, el botón de arranque del motor, el terminal y, por último, el personaje. A lo largo de este apartado se van mencionar la lógica y el diseño general que se necesita para cada uno de estos elementos, y se explicará el desarrollo y la implementación en el apartado siguiente.

4.2.1 Las baterías

Siempre que se quiera diseñar cualquier elemento jugable hay que especificar primero cuál es el comportamiento que espera el jugador sobre ese objeto. Para los objetos coleccionables, como pueden ser las baterías, el jugador espera que, al pasar sobre el objeto, este desaparezca, simulando de esta manera que el objeto ha sido recogido por el jugador. También se espera que el objeto emita un sonido que indique que el objeto se ha recogido.

A la hora de simular la recogida del objeto, este debe detectar que el jugador ha pasado a través de él. Para poder hacerlo es necesario añadir un componente de colisión [65].

Un componente de colisión, o una colisión, es un volumen de cualquier forma que es capaz de detectar cuando otro volumen u objeto ajeno choca o colisiona con él. Una vez se detecta este choque, pueden ocurrir tres situaciones en función del tipo de colisión que sea: las colisiones pueden bloquear, esto ocurre, por ejemplo, cuando un personaje se ve bloqueado por una pared, es decir, no puede atravesar la pared. Otra situación es que la colisión se ignore, es decir, el objeto atraviesa la colisión e ignora completamente su existencia. Por último, tenemos la superposición (*overlap*), esta situación es similar a ignorar pero en este caso, el motor nos informa de que esa colisión ha tenido lugar y que objeto ha realizado la colisión. Para nuestro caso particular, la colisión que debemos usar es esta última y capturar el evento de colisión para usar esa información más adelante.

En la Figura 4.2 se puede ver la representación final de la batería dentro del juego.

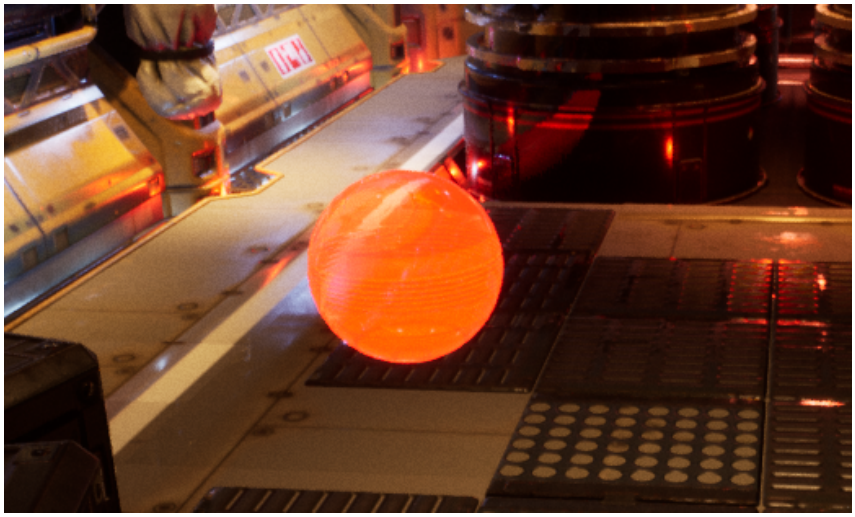


Figura 4.2: Diseño final de la batería

4.2.2 La puerta

Para el diseño de la puerta el comportamiento esperado es que, una vez el jugador obtenga las tres baterías, la puerta debe abrirse. Para ello, lo que se ha hecho es algo similar a lo que ocurre con las baterías. Se ha asociado una colisión a la puerta y, cuando el motor nos indica que ha ocurrido una superposición, se comprueba qué actor se ha superpuesto. En el caso de que esa superposición la haya realizado el jugador, se comprueba que el jugador ha recogido todas las baterías y la puerta se abre. En la Figura 4.3 se puede ver el diseño completo y la visualización cuando esta cerrada y abierta.

(a) Puerta cerrada (b) Puerta abierta

Figura 4.3: Diseño final de la puerta

4.2.3 El botón

Para el diseño del comportamiento del motor, podemos basarnos en el de la puerta. Al igual que con el diseño anterior, es necesario una colisión que detecte cuando el jugador ha entrado en el área de acción del botón. La comprobación esta vez, es un poco más compleja ya que no sólo se debe comprobar que el jugador está en rango y que ha pulsado la tecla de acción, si no que, además, la conexión con el servidor debe ser correcta. El diseño final del botón se puede ver en la Figura 4.4.

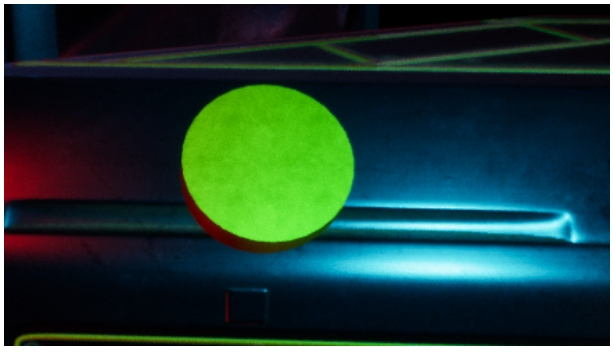


Figura 4.4: Diseño final del botón

4.2.4 El terminal

El terminal es la pieza clave de nuestro juego, es donde el jugador va a realizar todas las operaciones para conectarse al servidor. Para ello nuestro terminal debe capturar todas las teclas pulsadas, para que el juego no confunda las teclas pulsadas con *inputs* de movimiento. Una vez el jugador pulsa la tecla para acceder al terminal, el controlador del jugador cambia la posesión del jugador a la del terminal.

Mientras el controlador manda sobre el terminal, el jugador usará el teclado para introducir comandos y moverse por las diferentes pantallas del terminal. Para poder salir del terminal, bastará con pulsar la tecla de escape o tabulador. En la Figura 4.5 se puede ver el diseño visual y la localización del terminal.

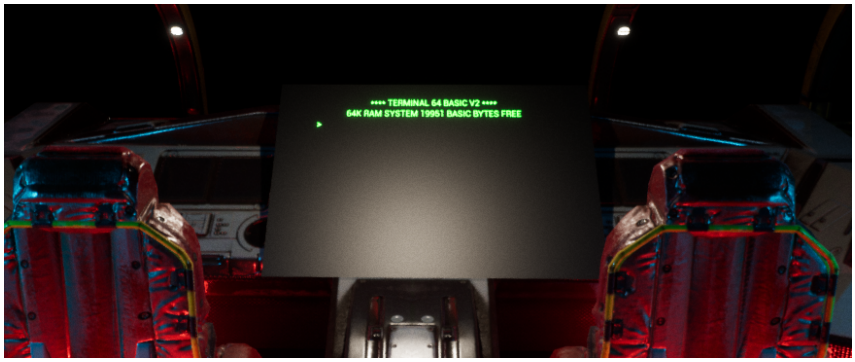


Figura 4.5: Diseño visual del terminal

4.2.5 El jugador

Por último, tenemos al jugador. El personaje del jugador no va a tener ningún modelo asociado a él y la cámara del personaje será en primera persona. Esto simplifica las cosas ya que no hay que modelar ni animar el avatar que representa al jugador.

Para el movimiento, el jugador pulsara las teclas 'a' y 'd' para moverse a izquierda y derecha respectivamente; las teclas 'w' y 's' para moverse hacia delante y hacia atrás. Para mirar a su alrededor podrá usar el ratón. Además de estos comandos básicos de movimiento, se va a implementar un menú de pausa donde el jugador podrá ver las misiones que le quedan por cumplir, así como los movimientos básicos. Para abrir y cerrar este menú bastará con pulsar la tecla *'esc'* o *'tab'*.

Asimismo, el jugador podrá pulsar la tecla 'f' para poder pulsar el botón de la nave y la tecla 'e' para acceder al terminal. Estas dos acciones son contextuales ya que solo se llevarán a cabo si el jugador se encuentra dentro del rango efectivo.

4.3 Diseño del terminal

En los apartados anteriores hemos explicado brevemente el funcionamiento general que se espera por parte de cada elemento básico para nuestro juego. Sin embargo, a la hora de hablar del terminal solo hemos comentado cuál es la interacción con el jugador, no hemos hablado del comportamiento ni de la estructura interna, es decir, qué pasos debe seguir el jugador y qué comandos debe usar para poder conectarse con el servidor.

El terminal consta de cinco carpetas. Dentro de cada carpeta se pueden ejecutar ciertos comandos específicos. En la Figura 4.6 se puede ver la estructura básica de los comandos. Los comandos están

representados con elipses y las carpetas con rectángulos. Cada carpeta dentro de *MainFolder* representa diferentes niveles de la estructura de redes, de los cuales solo se han representado las capas que se van a usar para la configuración especifica en nuestro juego. Además, dentro de la carpeta principal se encuentra el comando de *status* que indica al jugador cual es la información especifica de la configuración realizada. Por último, tenemos el comando de ayuda, indicado con el signo de interrogación, este comando es válido a cualquier nivel e imprime los comandos válidos de cada carpeta, así como una breve explicación de cada uno de ellos.

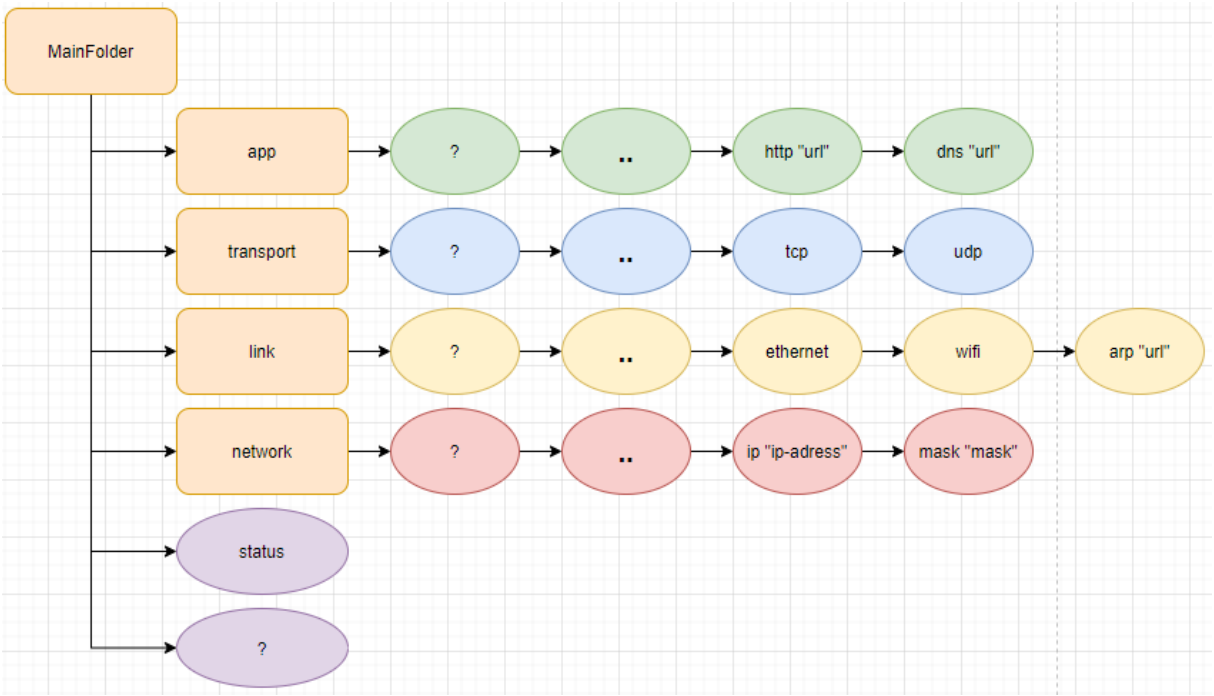


Figura 4.6: Estructura de carpetas y comandos del terminal

Dentro de cada carpeta hay dos comandos comunes, el primero es el de ayuda. Como se ha comentado anteriormente, imprime por pantalla la información de los comandos válidos de cada carpeta. El otro comando común es el comando para volver a la carpeta principal, para ejecutarlo tan solo hay que introducir dos puntos en la barra de comandos. La primera carpeta, app, posee dos comandos. El primero de ellos es *dns 'url'*, este comando permite al jugador obtener la información de la IP y de la máscara de la URL especificada. El segundo comando (*http 'url'*) sirve para realizar la conexión con el servidor en base a la configuración especificada por el jugador.

Para la carpeta de *transport* se tienen dos comandos que establecen el protocolo de la capa de *transport*, en nuestro caso se elige entre los protocolos tcp o udp. De forma similar ocurre en la capa de *link*, se elige entre conexión *ethernet* y *wifi*. Además de estos dos comandos, en la carpeta de *link* se tiene otro comando (*arp 'url'*) que sirve para mostrar y almacenar en la configuración la MAC del servidor al que nos queremos conectar. Por último, tenemos la carpeta de *network*, en ella se configura la IP y la máscara del propio terminal.

Los pasos básicos que debe seguir el jugador para poder realizar la conexión son los siguientes:

1. El jugador debe obtener la IP y la máscara del servidor al que quiere acceder por lo que deberá introducir el comando *dns 'url'* dentro de la carpeta de app, el terminal le devolverá como respuesta la máscara y la IP del servidor, pero no almacenará esa información en ningún lado. Será trabajo del jugador recordar y especificar más adelante la IP y máscara a utilizar por parte del terminal.

2. Una vez se tiene la IP y la máscara, el jugador deberá establecer la ip y la máscara del cliente teniendo en cuenta las restricciones de la misma. Esta configuración se realiza en la carpeta de network.

3. Una vez realizada la configuración de IP y máscara, es necesario especificar el protocolo de la capa de transporte, el jugador deberá introducir el tipo de protocolo en la carpeta *transport*.

4. Para terminar, hay que especificar el tipo de enlace del servidor, es decir, ethernet o wifi. Esto se realiza en la carpeta *link*. Dentro de esta misma carpeta se debe usar el comando *arp* para almacenar la dirección MAC del servidor, en este caso con tan solo ejecutar el comando se almacenará automáticamente en la configuración.

5. Si se han seguido los pasos correctamente y se han realizado las configuraciones adecuadas, tan solo queda realizar la petición de conexión. El jugador deberá situarse en la carpeta de app y realizar la petición usando el comando http. Este comando indicará si la conexión se ha realizado adecuadamente.

4.4 Mapa y entorno

Para poder comenzar con el desarrollo y la implementación del proyecto dentro de Unreal, es necesario crear un entorno 3D que se adecue a las características que se necesitan para el proyecto. Crear un entorno virtual desde cero lleva mucho trabajo, incluso si se tienen todos los *assets*. Como el objetivo de este proyecto no es la creación de un mapa, se va a usar una serie de *assets* modulares que se encuentran de forma gratuita dentro del bazar de la *Epic Store* (ver Figura 4.7). Estos *assets* vienen con un mapa de demostración que es el que se va a usar y modificar para este TFG. Este mapa, así como los *assets*, han sido creados y diseñados por Denys Rutkovskyi.

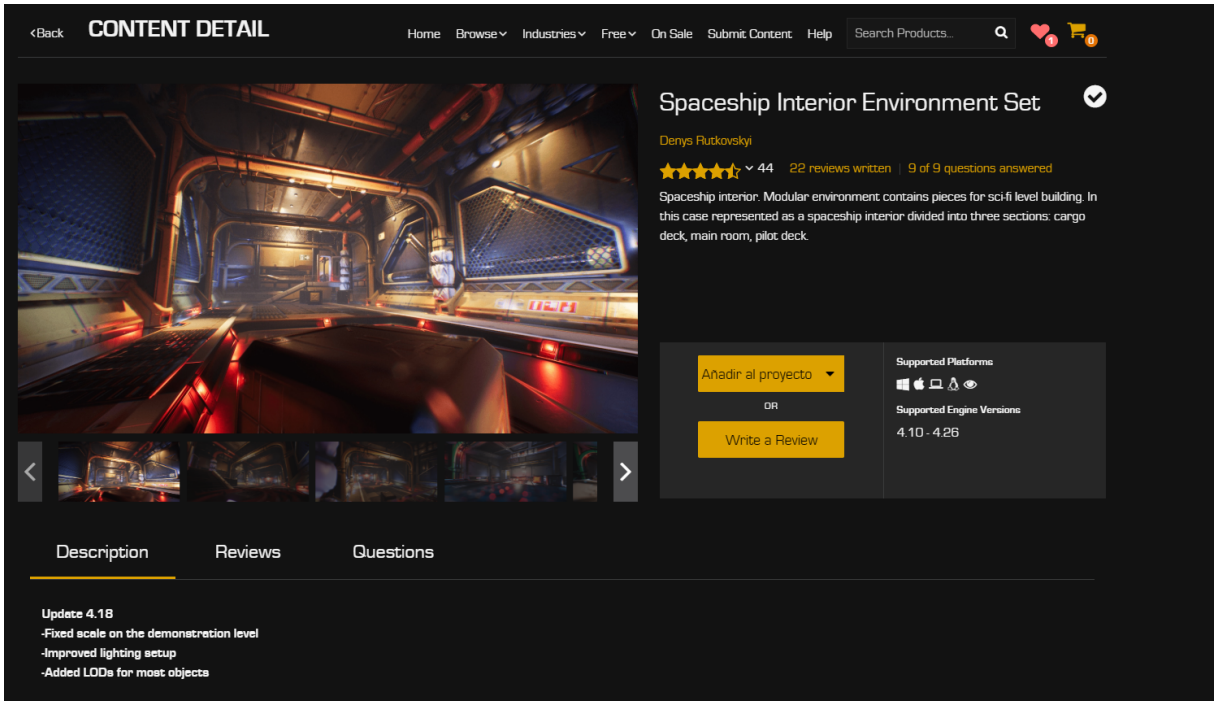


Figura 4.7: Página de nuestro mapa en la *Epic Store*

Debido a las características del mapa, este es demasiado pequeño para lo que queremos realizar, ya que tan solo consta de tres salas: la sala de control, una sala intermedia y una cubierta de carga. El plano del mapa se puede ver en la Figura 4.8. Para añadir elementos de exploración básica se ha ampliado el mapa incluyendo dos salas adicionales anexionadas a los laterales de la sala intermedia, tal y como se puede ver en la Figura 4.9. Los marcadores rojos representan la localización de las baterías que el jugador debe coleccionar para poder acceder a la sala de control. Dentro de la sala de control se localizan el botón de arranque (círculo azul) y la terminal de acceso. Por último, en el centro del mapa, el muñeco representa el lugar de inicio del jugador.

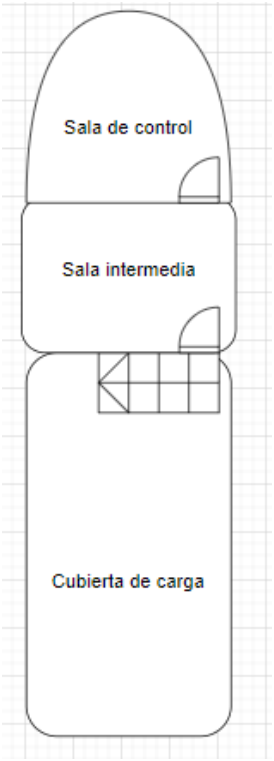


Figura 4.8: Plano mapa original

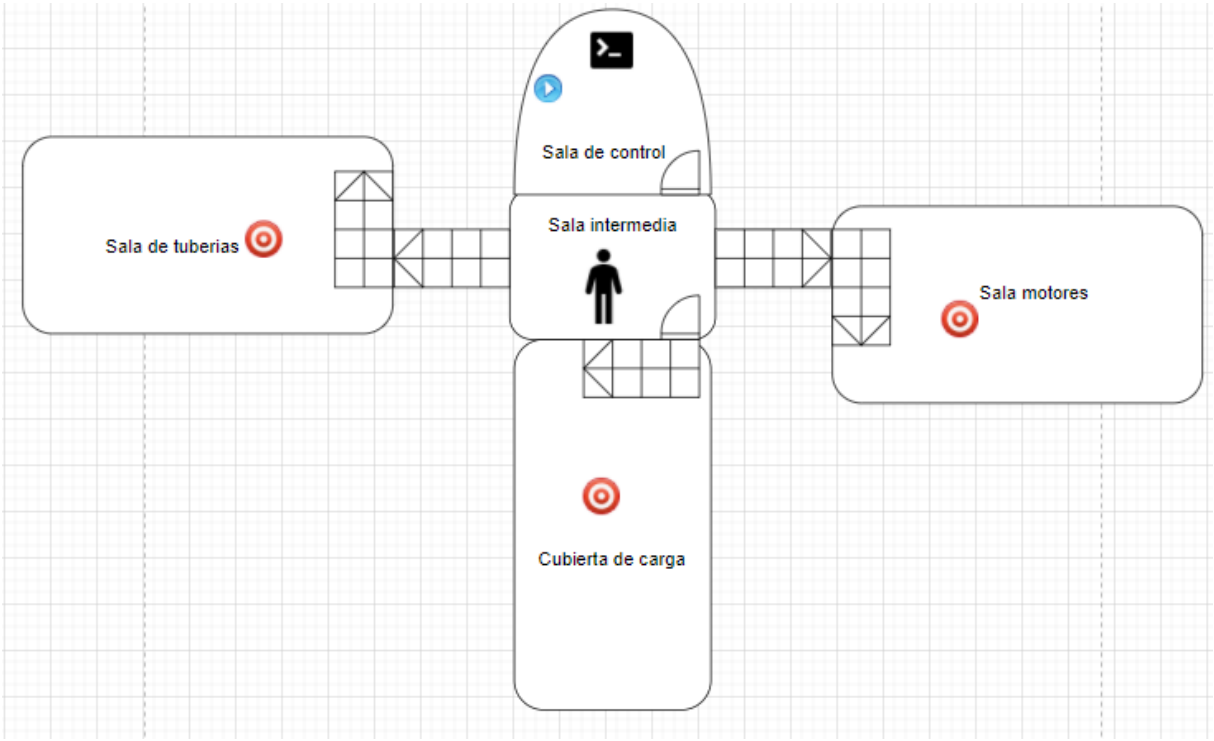


Figura 4.9: Mapa final

Capítulo 5

Implementación

Durante el siguiente apartado se va a explicar la implementación de los diferentes elementos que conforman el proyecto completo. Se hablará del desarrollo técnico del mismo basándonos en el comportamiento definido durante la fase de diseño.

5.1 Implementación del personaje

El primer paso a seguir una vez hemos realizado el mapa, es la creación de un personaje que pueda moverse e interactuar en el mismo. Para ello se ha creado una clase de C++ con los *inputs* básicos, según se ha especificado en el apartado anterior. Estos *inputs* básicos son los que van a permitir al jugador explorar el mundo y realizar acciones básica en este. Esto incluye que el jugador pueda moverse y mirar a su alrededor, así como los botones de interacción para pulsar el botón y acceder al terminal.

Para poder establecer los *inputs* debemos ir a la configuración del proyecto dentro de Unreal. En la pestaña de *input* se especifican el nombre de la acción y la tecla que va asociada. En la Figura 5.1 se puede ver una parte de la configuración de movimiento básica que se ha realizado para este proyecto. Como se puede ver, es posible asignar diferentes teclas e *inputs* a una misma acción. Este mapeado de teclas permite añadir un nivel de abstracción que permite una mayor flexibilidad para trabajar. Por ejemplo, si se toma la decisión de modificar los *inputs* finales del proyecto la tarea de modificarlo resulta más sencilla ya que tan solo hay que modificar los *inputs* en las preferencias del proyecto.

A la hora de generar y enlazar los eventos de *input* Unreal diferencia dos dos posibilidades: mapas de acción y mapas de eje [66]. Los primeros se usan cuando solo se necesitan los eventos de tecla presionada y tecla liberada, mientras que los mapas de eje permiten que el *input* sea de rango continuo.

Una vez se han establecido los *input* básicos, se deben unir a la acción especifica que se quiere realizar al propio *input*. En la Figura 5.2 se puede ver el código de la clase base del jugador, denominado *PlayerCharacter.cpp*. Esta clase deriva de la clase *Character* de Unreal y nos permite llamar a funciones especificas de esta clase. Según se ve en la función *SetupPlayerInputComponent*, se asigna una función al *input* que hemos definido. Por ejemplo, el *input* definido como *MoveForward* tiene asociado la función *MoveForward*, que está definida unas lineas más abajo. Esta función llama a la función *AddMovementInput* (heredada de la clase *Character*), esto permite avanzar la posición del jugador hacia delante en base al vector de posición asociado a él.

Para asignar los *inputs* a las acciones llamamos a la función *BindAxis* o *BindAction*, en función del tipo de *input* que queramos. Los atributos de ambas funciones son un *String* indicando el nombre asignado

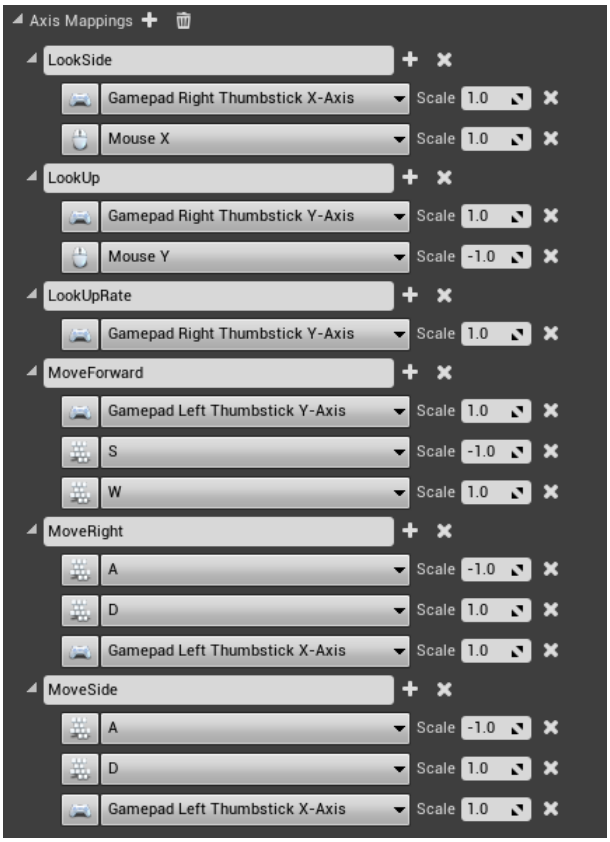


Figura 5.1: Configuración de las teclas

al *input* que se ha definido en las preferencias del proyecto, el tipo de evento que tiene que ocurrir para que se ejecute la función, el objeto usuario al que se va a enlazar el *input* y la función a ejecutar. Para el caso de *BindAxis* no hay que especificar el evento de *input*. Ambas funciones derivan del componente de *input* [67] asociado directamente a la clase de *character*. La definición de estas funciones se pueden ver en las figuras 5.4 y 5.3.

Además de haber relacionado los *inputs* con su correspondiente función, queda una cosa más por hacer. En la Figura 5.5 se puede ver parte del archivo *PlayerCharacter.h*, en este archivo hemos definido dos variables adicionales que nos van a permitir llevar el recuento de las baterías que el jugador ha ido recogiendo a lo largo de juego. Estas variables son *NumberOfBateries* que nos indica el numero total de baterías que el jugador debe recolectar, y la variable *BateriesCollected* que indica el numero de baterías que el jugador ha recogido. En la linea inmediatamente superior a la declaración de la variable se puede ver unos atributos asociados a ella que permite la modificación de esas variables dentro de los *Blueprints* del juego sin necesidad de volver a recompilar todo el código.

Una vez se han desarrollado las bases de la lógica del personaje, creamos una nueva clase *Blueprint* que derive de esta, llamada *BP_PlayerCharacter*. Esta será la clase con la que iremos trabajando y desarrollando nuestro proyecto. Al usar un *Blueprint* podemos realizar pruebas mucho más rápido que si trabajásemos directamente con la clase de C++, ya que evitamos los largos tiempos de compilación.

5.2 Implementación de las baterías

Teniendo en cuenta las especificaciones de diseño que se han explicado en el apartado anterior, para poder diseñar la baterías hay que añadir una colisión. En la Figura 5.6 se pueden ver los componentes

PlayerCharacter.h M PlayerCharacter.cpp M

EscapeTheShip > Source > EscapeTheShip > PlayerCharacter.cpp > APlayerCharacter::SetupPlayerInputComponent(UInputComponent *)

```cpp
// Fill out your copyright notice in the Description page of Project Settings.
#include "PlayerCharacter.h"
// Sets default values
APlayerCharacter::APlayerCharacter()
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
}
// Called when the game starts or when spawned
void APlayerCharacter::BeginPlay()
{
    Super::BeginPlay();
}
// Called every frame
void APlayerCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}
// Called to bind functionality to input
void APlayerCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);

    PlayerInputComponent->BindAxis(TEXT("MoveForward"), this,&APlayerCharacter::MoveForward);
    PlayerInputComponent->BindAxis(TEXT("MoveSide"), this,&APlayerCharacter::MoveSide);
    PlayerInputComponent->BindAxis(TEXT("LookUp"),this,&APawn::AddControllerPitchInput);
    PlayerInputComponent->BindAxis(TEXT("LookSide"),this,&APawn::AddControllerYawInput);
    PlayerInputComponent->BindAction(TEXT("Jump"),EInputEvent::IE_Pressed,this,&ACharacter::Jump);
}

void APlayerCharacter::MoveForward(float AxisValue)
{
    AddMovementInput(GetActorForwardVector() * AxisValue);
}

void APlayerCharacter::MoveSide(float AxisValue)
{
    AddMovementInput(GetActorRightVector() * AxisValue);
}
```

Figura 5.2: Código PlayerCharacter.cpp

```cpp
template<class UserClass>
FInputAxisBinding & BindAxis
(
    const FName AxisName,
    UserClass * Object,
    typename FInputAxisHandlerSignature::TUObjectMethodDelegate< UserClass >::FMethodPtr Func
)
```

Figura 5.3: Documentación bindAxis

```cpp
template<class UserClass>
FInputActionBinding & BindAction
(
    const FName ActionName,
    const EInputEvent KeyEvent,
    UserClass * Object,
    typename FInputActionHandlerWithKeySignature::TUObjectMethodDelegate< UserClass >::FMethodPtr Func
)
```

Figura 5.4: Documentación bindAction

```
private:
    void MoveForward(float AxisValue);
    void MoveSide(float AxisValue);
    UPROPERTY(EditAnywhere,BlueprintReadWrite, Category = "Objects", meta = (AllowPrivateAccess = "true"))
    int NumberOfBateries = 3;
    UPROPERTY(EditAnywhere,BlueprintReadWrite, Category = "Objects", meta = (AllowPrivateAccess = "true"))
    int BateriesCollected = 0;
```

Figura 5.5: Código PlayerCharacter.h

asociados a las baterías, así como su representación visual en el mundo. En la columna de la izquierda se diferencian tres componentes que se organizan en base a un árbol de prioridades. El primer componente es *DefaultSceneRoot*, este componente hereda de la clase Actor y define la transformada (localización, rotación y escala) del objeto. Todos los componentes adicionales deben estar asociados a este. Se representa visualmente con una esfera de color blanco. El siguiente componente es un volumen esférico (*sphere*) al cual se le ha asignado un material translúcido de color rojo. Por último, tenemos la colisión. Para poder visualizarla se colorean los limites del volumen de color rojo, en nuestro caso el tamaño el ligeramente superior a la esfera roja.

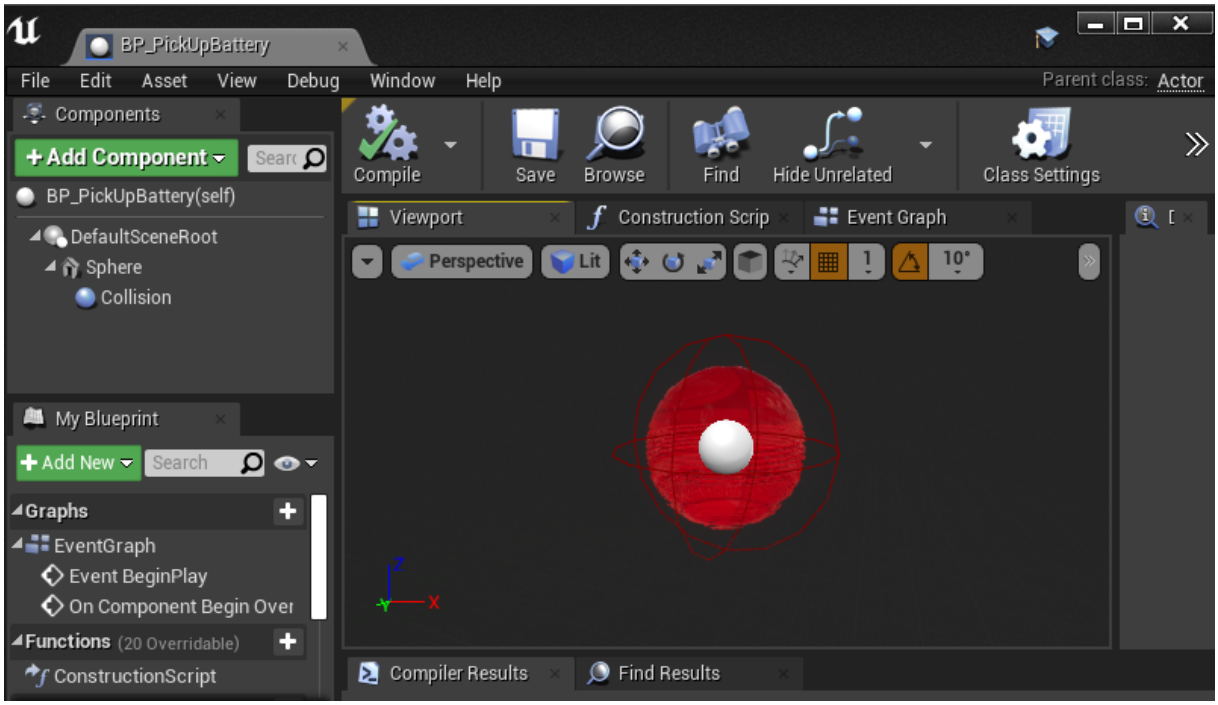


Figura 5.6: Componentes de BP_PickUpBattery

Una vez se han establecido los componentes principales de nuestro objeto, pasamos a la programación de la lógica. Toda esta programación se ha realizado en *Blueprints* y toda esta lógica se puede ver en la Figura 5.7.

Para poder programar la lógica vamos a crear la función *OnComponentBeginOverlap*, esta función se va a llamar cada ver que un actor se superponga con la colisión, una vez se ha producido el evento llamamos a una función que nos devuelve el jugador y sumamos en una unidad la variable de las baterías recolectadas. A continuación obtenemos la localización del jugador y emitimos un sonido. Por último, destruimos el actor, simulando así la recolección del objeto.

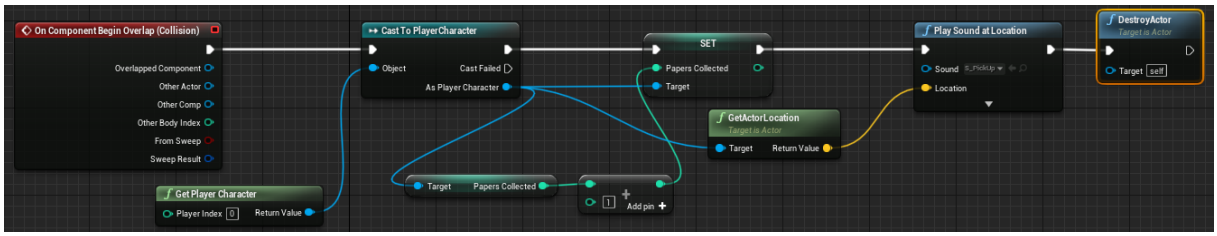


Figura 5.7: Código de BP_PickUpBattery

5.3 Implementación de la puerta

Para la implementación de la puerta seguimos el mismo proceso que para las baterías, en la Figura 5.8 se puede ver que tiene la colisión denominada *triggerVol*. Además de la colisión, para este caso particular, el objeto de la puerta consta de dos *assets* que son el marco de la puerta y la puerta en sí.

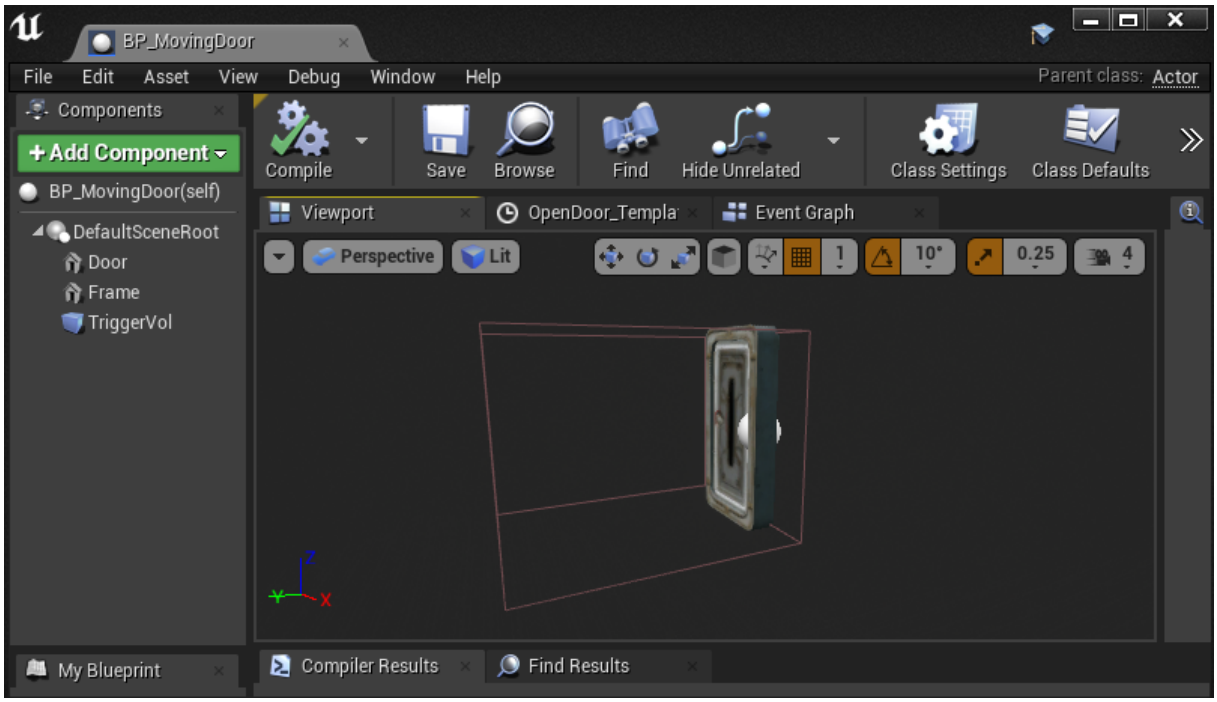


Figura 5.8: Componentes de BP_MovingDoor

A la hora de definir el comportamiento de la puerta se sigue el mismo proceso que con las baterías. Una vez el jugador atraviesa la colisión, se comprueba que este posea el numero de baterías necesarias para abrir la puerta, si se cumple la condición (nodo *branch*) se activa la animación de apertura. Se establece un temporizador de 1 segundo (nodo *OpenDoor*) que realiza una interpolación(ver Figura 5.9) en base a una función. Para este caso la relación de interpolación es lineal por lo que la animación se realizara de forma constante a lo largo del tiempo. La función que realiza la animación para nuestro caso particular es la rotación relativa en el eje z de la puerta (nodo *setRelativeRotacion*). Una vez ha terminado la animación es necesario destruir el componente de la colisión para que no se vuelva a activar si el jugador decide volver a pasar por la puerta.

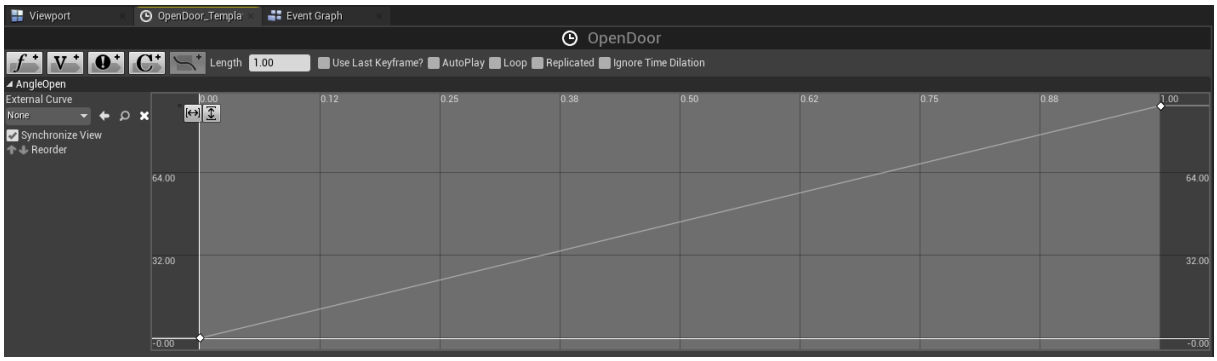


Figura 5.9: Linea temporal de la animación

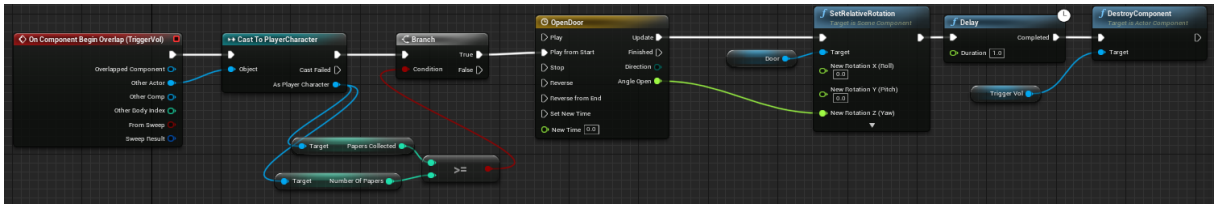


Figura 5.10: Código de BP_MovingDoor

5.4 Implementación del botón de arranque

Al igual que ocurre con la implementación de la puerta y las baterías, el botón necesita de una colisión en sus componentes, tal y como se ve en la Figura 5.11.

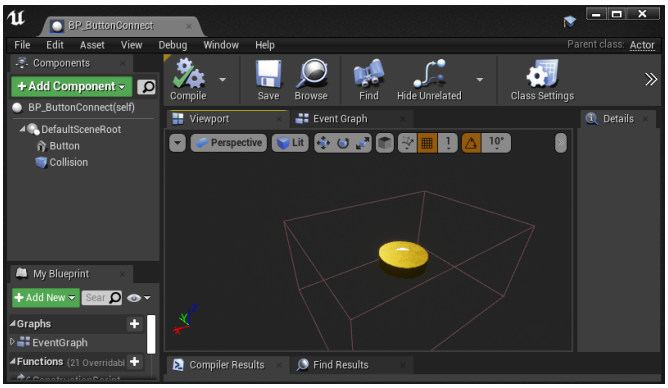


Figura 5.11: Componentes de BP_ButtonConnect

Para la programación de la lógica, se usa la misma función que para la puerta y las baterías. Para este caso concreto se van a usar dos funciones. La primera para detectar cuándo el jugador ha entrado dentro del rango de la colisión y otra para detectar cuándo el jugador ha salido de ese rango.

Cuando se detecta que el jugador ha entrado dentro de ese rango se activa una variable booleana que se ha creado que indica si el jugador se encuentra dentro del rango. Además, se crea un *widget Blueprint* [68] que muestra en la pantalla del jugador un texto indicándole qué tecla debe pulsar para poder arrancar el motor. Un *widget Blueprint* es una interfaz visual que sirve para crear menús y cuadros de texto en la pantalla del jugador. Para este tipo de componentes se usa la herramienta de Unreal UMG (*Unreal motion Graphics*) [69].

Si el jugador sale del rango, se llama a la función *OnComponentEndOverlap* que marca la variable del rango a falso y elimina el *widget* de la vista del jugador. Todo este código se puede ver en la Figura 5.12.

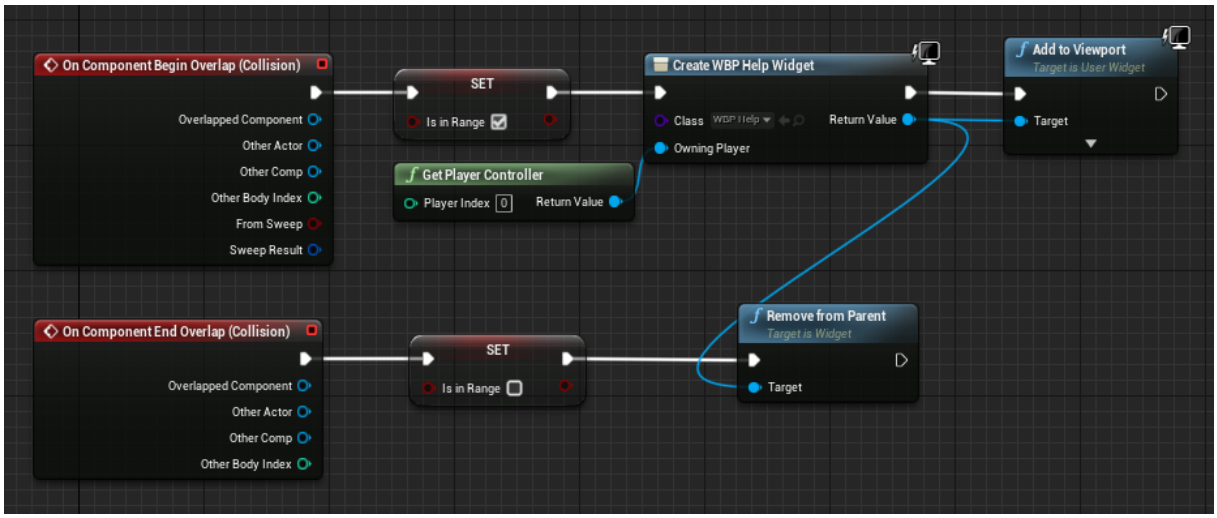


Figura 5.12: Código de BP_ButtonConnect

5.5 Implementación del terminal

A la hora de programar el terminal hay que crear varios objetos (actores) y *Blueprints* conectados entre sí. Para ello vamos a basar nuestro terminal en uno ya existente [70] dentro del *marketplace* de Unreal. Este terminal nos viene con la implementación de la transición de la cámara y la captura de las teclas.

Para poder comprender el funcionamiento y la lógica de ejecución de este terminal, primero hay que explicar ciertos conceptos básicos. Este terminal se basa en una interfaz de usuario (*widget* [68]), esta interfaz se modifica cada vez que el jugador introduce un *input* por teclado. Para que esto sea posible es necesario que el controlador del jugador deje de controlar al jugador predeterminado y pase a controlar el terminal.

Al tratarse de un sistema de interfaz de usuario, es necesaria una conexión entre el objeto 3D y la interfaz que se va mostrando. Además, el sistema debe detectar cuándo se está usando la terminal y cuándo no. Para poder hacer la implementación se va a crear un *Blueprint* que sera el terminal físico dentro del juego, después se debe crear un objeto derivado de la clase Actor, que conectará el terminal físico con las interfaces. Por último, se van a crear tres interfaces, que serán las encargadas de procesar y mostrar los comandos del jugador y sus respuestas.

5.5.1 *BP_TerminalSystem*

Este objeto es la terminal física con la que interactúa el jugador, es la parte visible del terminal y la que agrupa el resto de elementos necesarios para que este funcione correctamente. En la Figura 5.13 se puede ver la estructura y los componentes que conforman esta clase.

5.5.2 *BP_TerminalStream*

Este objeto es la conexión entre la interfaz y el terminal físico. Esta asociado como variable en el terminal físico. Se encarga de actualizar la visualización del *widget* cada vez que se produzca un cambio. Para que esto sea posible es necesario añadir un componente denominado *SceneCaptureComponent2D* [71], este componente necesita de una cámara que permita *renderizar* el plano. En la Figura 5.14 se puede ver la estructura y organización de los componentes.

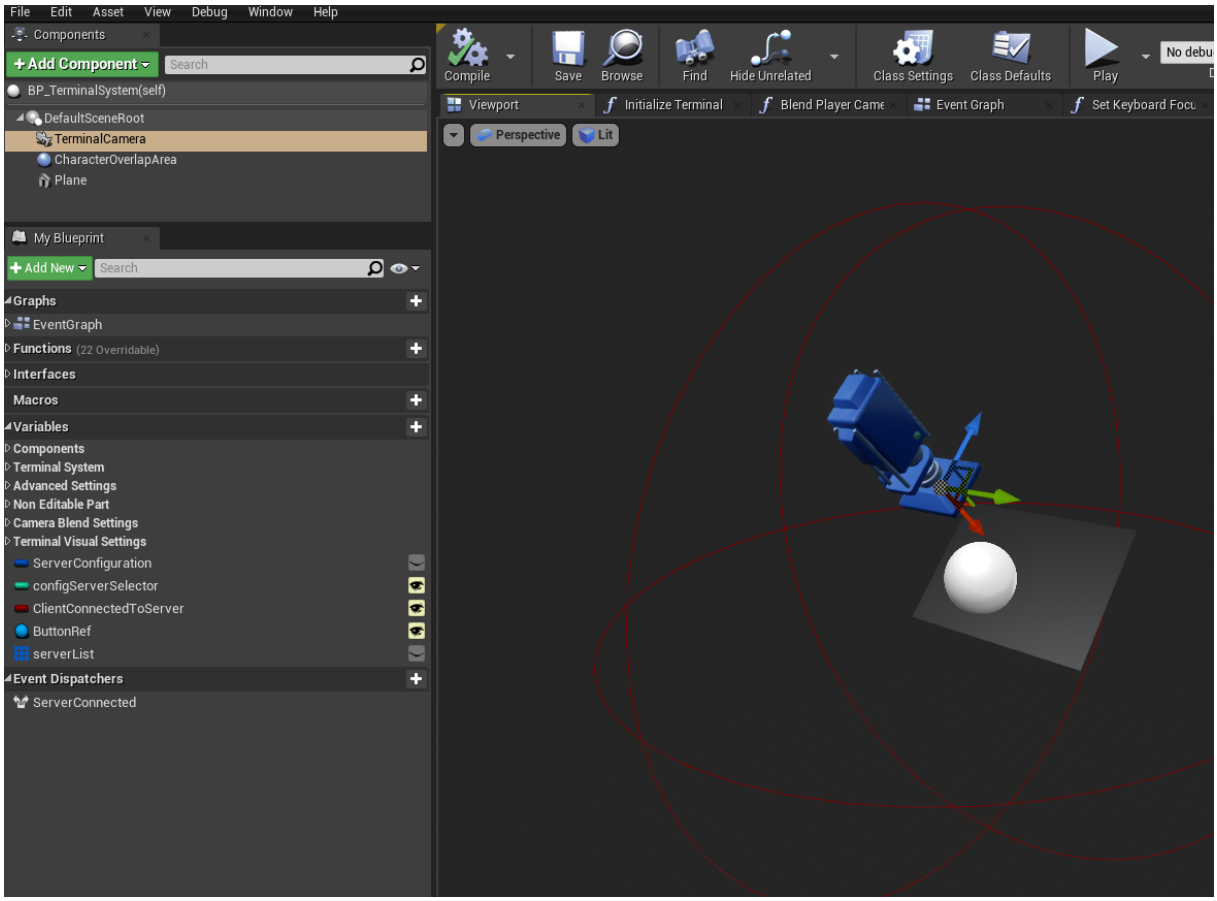


Figura 5.13: Componentes y visualización de BP_TerminalSystem

Este objeto consta de la función *InitializaTerminalWidget* indica al componente de *render* que *widget* debe inicializarse y que terminal físico debe establecerse para la relación. Esta función se llama al iniciar el juego y no varia a lo largo de él. En la Figura 5.15 se puede ver la lógica de la función.

5.5.3 *WBP_TerminalCommandLine*

Este *widget* sirve para modificar y visualizar el texto del *widget* principal, así como para detectar y almacenar los valores de las teclas pulsadas. En la Figura 5.16 se puede ver que este *widget* se conforma de cuatro componentes, los tres primeros componentes (*canvasPanel*, *HorizontalBox* y *sizeBox*) sirve para ajustar el tamaño y la posición del texto. El último componente, identificado con el nombre *CommandLine* se trata de un *editable text multiline* [72], es decir, un editor de texto multilínea, este componente funciona como un campo de *input* que detecta automáticamente las teclas pulsadas y las almacena en la variable *commandLine*.

Para poder gestionar la variable de *commandLine* y saber cuando enviar al terminal, se ha creado una función que se ejecuta antes de añadir la tecla al terminal. El código de esta función se puede ver en las Figuras 5.17, 5.18, 5.19 y 5.20 . Como atributos de entrada de la función tenemos la variable *InKeyEvent*, que indica qué tecla se ha pulsado. Lo primero que se hace es comprobar si la tecla pulsada es la tecla *enter*, si se cumple esta condición (ver Figura 5.17), se envía el texto al *widget* de *WBP_TerminalMain*, que es la clase encargada de procesar el comando. Si no se cumple esta condición, se comprueba si el jugador ha pulsado la tecla de salida del terminal.

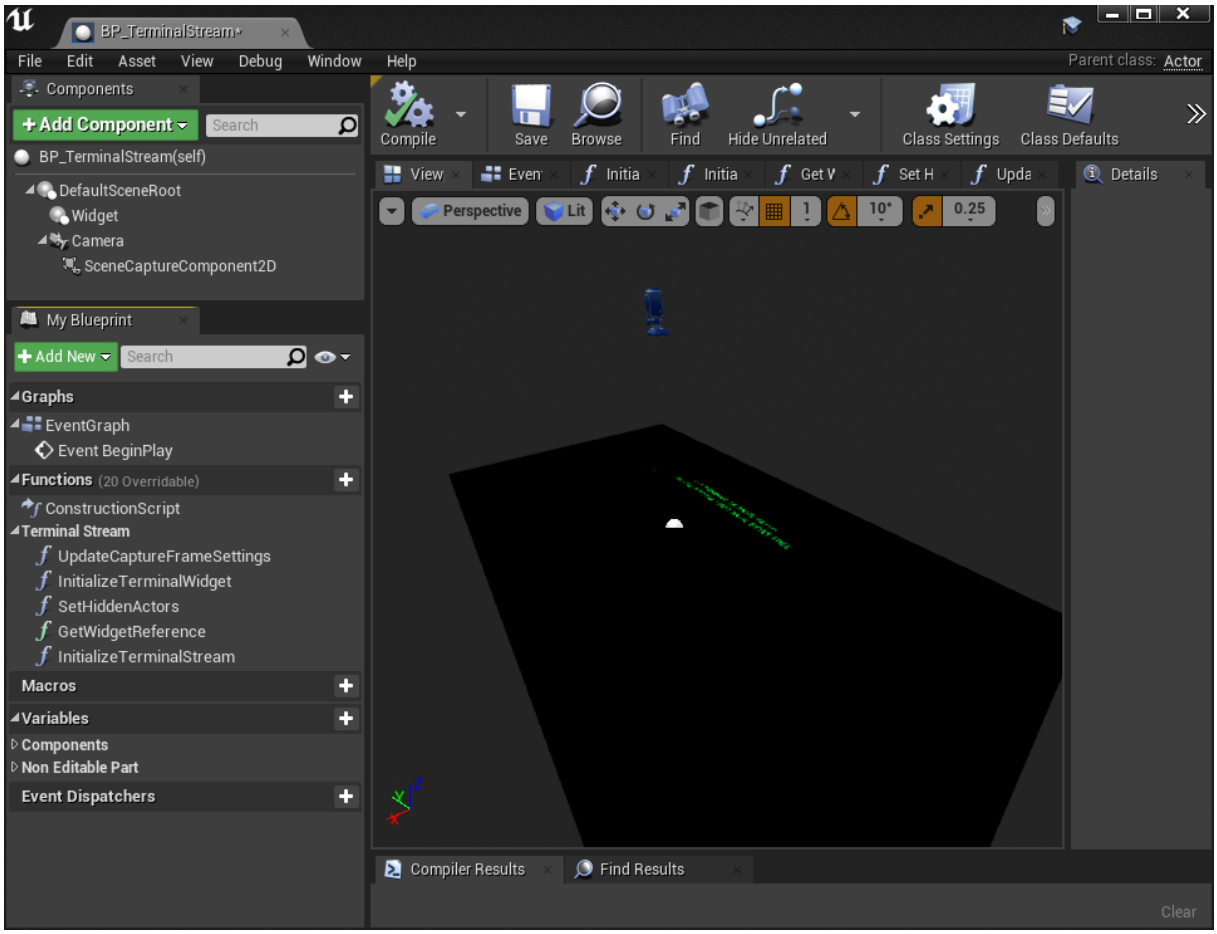


Figura 5.14: Componentes y visualización de BP_TerminalSystem

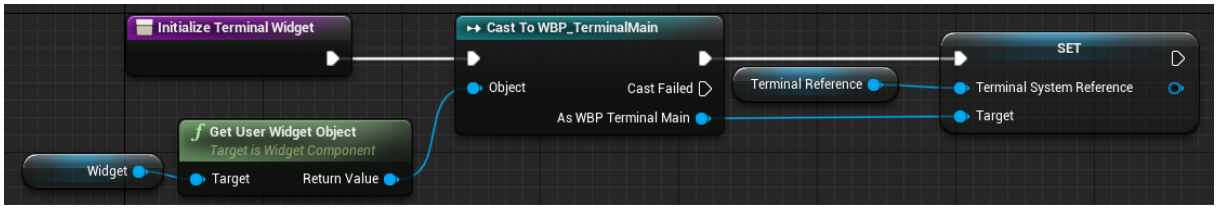


Figura 5.15: Código de la función *InitializaTerminalWidget*

Como se ha comentado antes, si el jugador pulsa la tecla *enter*, se envía el texto guardado al *WBP_TerminalMain*, para poder hacer esto transforma el texto de la variable *CommandLine* a formato *string*. Una vez transformado se añade esta variable a la linea de comandos con una función específica de esta clase. El código de esta clase se puede ver en la Figura 5.18.

Si el jugador no ha pulsado la tecla *enter* se comprueba si la tecla pulsada, coincide con la pulsada para salir del terminal. Si se cumple la condición, usando la referencia a la clase *BP_TerminalSystem* se comprueba si es posible realizar la transición del terminal al jugador. Si se puede realiza esta transición. Para poder realizar este cambio de forma adecuada es importante establecer el modo de *input* a solo juego (usando la función *setInputModeGameOnly* [73]. El código de esta función se puede ver en las Figuras 5.19 y 5.20.

Figura 5.16: Diseño y jerarquía de componentes de la clase *WBP_TerminalCommandLine*

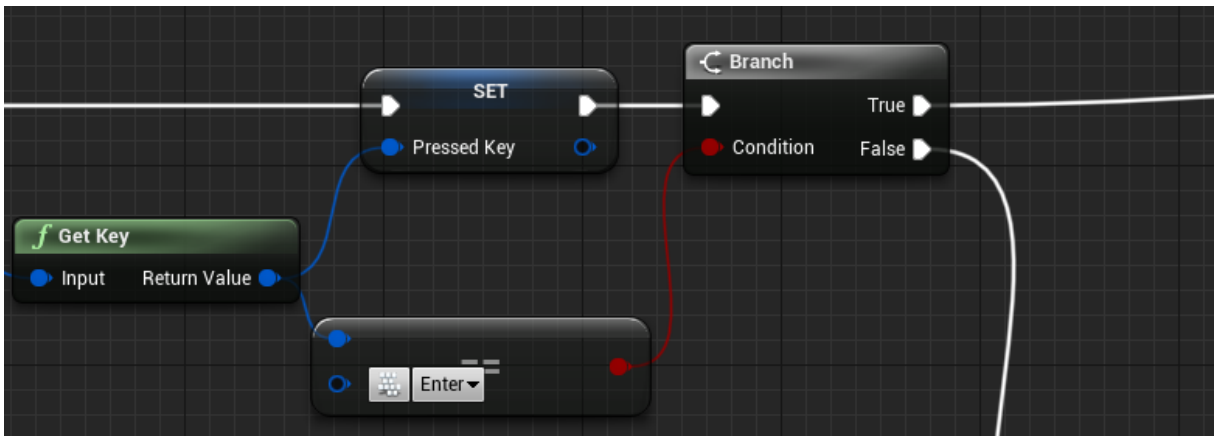


Figura 5.17: Comprobación de la tecla pulsada. Función *OnPreviewKeyDown* parte A

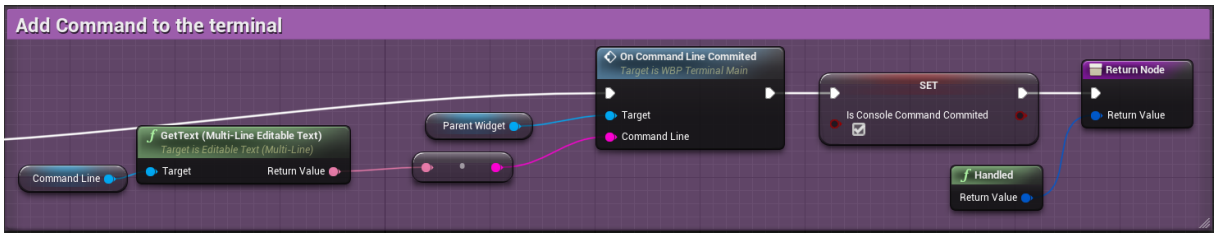


Figura 5.18: Código que introduce el comando en la línea del terminal. Función *OnPreviewKeyDown* parte B

5.5.4 *WBP_TerminalScreenMessagge*

Este es el *widget* más sencillo de los tres que se han implementado para este sistema. Este *widget* está conectado con el terminal físico mediante el *BP_terminalStream* y muestra un texto en pantalla indicándole al jugador qué tecla debe pulsar para acceder al terminal en cuestión. En la Figura 5.21 se ve un ejemplo de cómo visualiza el jugador este *widget*.

Este *widget* se activa cada vez que el jugador entra dentro del rango efectivo del terminal y se desactiva al salir el jugador del mismo. El código de este *widget* (ver Figuras 5.22 y 5.23) comprueba inicialmente

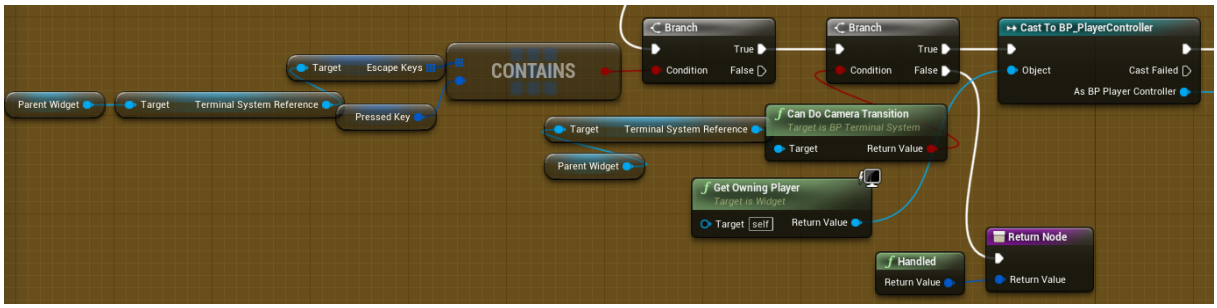


Figura 5.19: Función *OnPreviewKeyDown* parte C

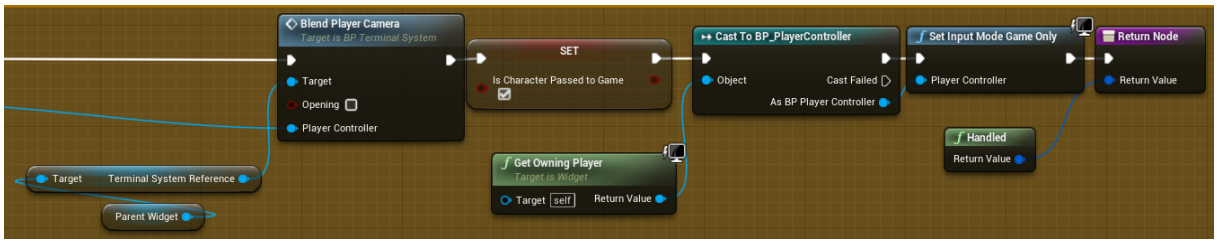


Figura 5.20: Función *OnPreviewKeyDown* parte D

Figura 5.21: Ejemplo de visualización de *InitializeDisplayWidget*

que tiene asociado un terminal. Después, se obtiene una referencia a él y comprueba que este terminal físico tiene una referencia al jugador. A continuación, obtiene el *input* que permite la transición de jugador a terminal y lo muestra por pantalla.

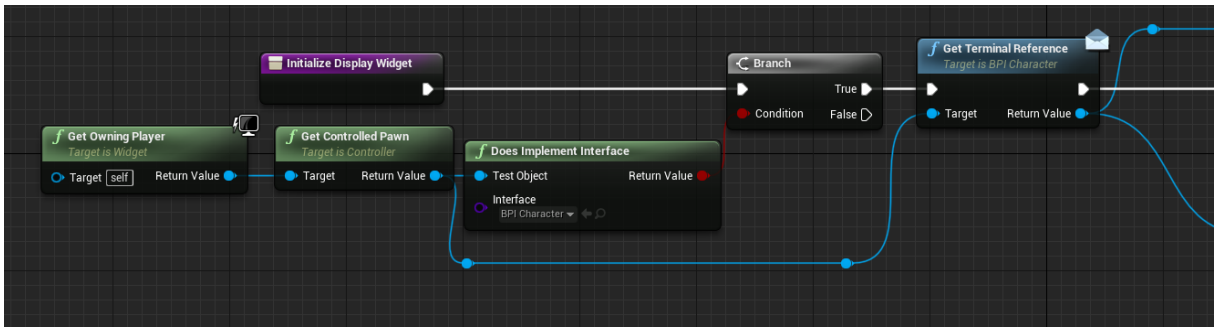


Figura 5.22: Código de la función *InitializaDisplayWidget* (parte A)

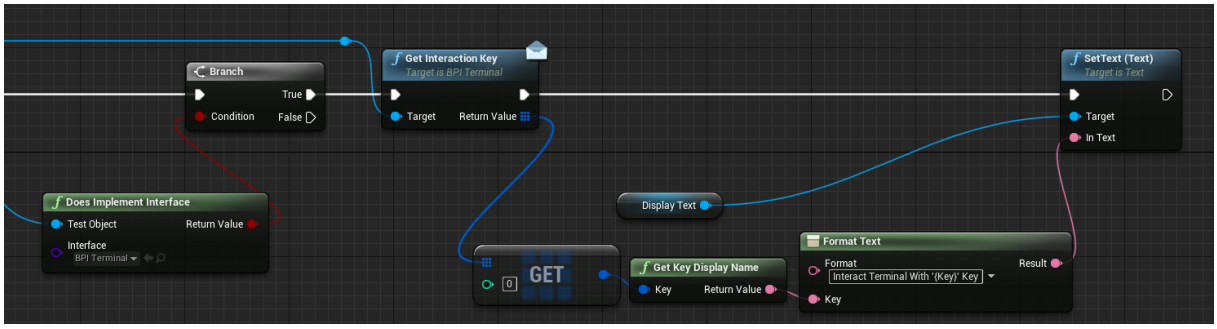


Figura 5.23: Código de la función *InitializaDisplayWidget* (parte B)

5.5.5 *WBP_TerminalMain*

Este *widget* es el encargado de procesar los comandos para emitir la respuesta y enviárselo al *WBP_TerminalCommandLine*. En la Figura 5.24 se puede ver el diseño visual de la interfaz.

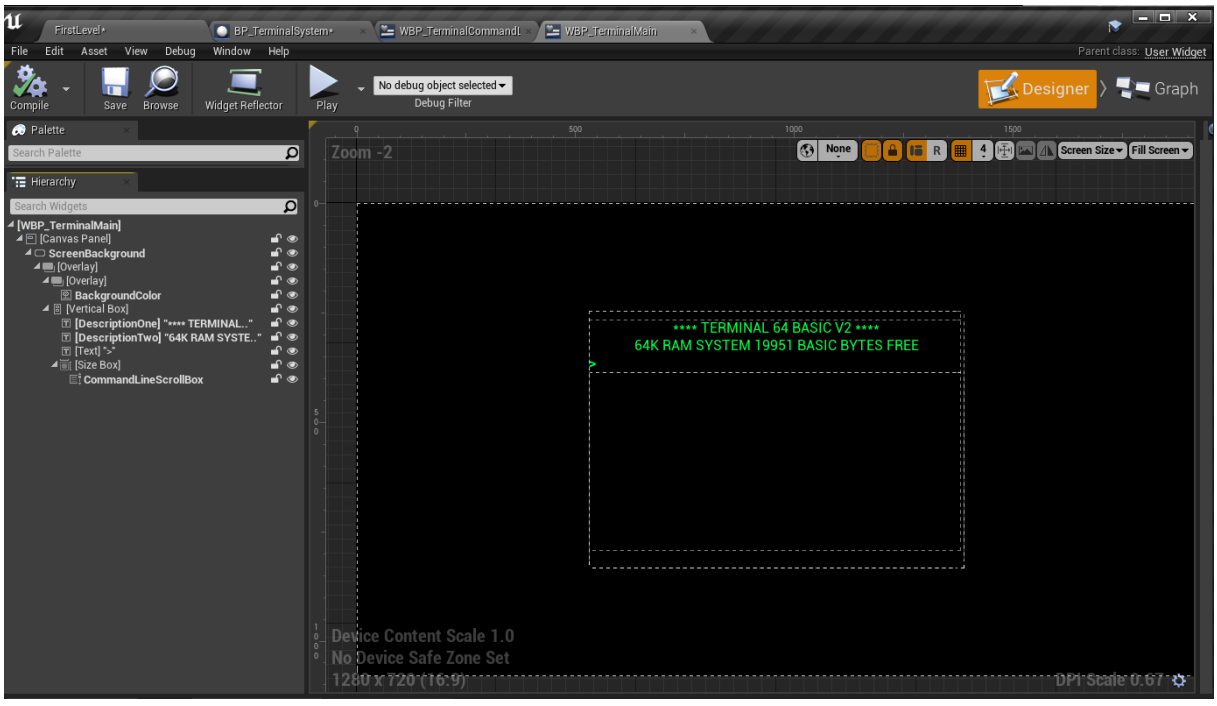


Figura 5.24: Diseño visual de *WBP_TerminalMain*

Antes de poder implementar la funcionalidad especifica para nuestro proyecto, hay dos funciones que se deben implementar. La primera de ellas se encarga de procesar el comando que el jugador escribe. Para ello se ha creado una función denominada *OnCommandLineCommited*. El código de esta función se puede ver el la Figura 5.25. El código recibe como parámetros de entrada un *string*, identificado con el nombre *commandLine*, este *string* se añade a un *array* que almacena todos los comandos enviados por el jugador, a continuación se procesa el comando del jugador y se añade una nueva linea de comando. La segunda función es la de añadir una respuesta al terminal de comando. Para ello se crea un nuevo widget de la clase *WBP_TerminalCommandLine* y lo añade al componente *CommandLineScrollBox*, el código de esta función puede ver el la Figura 5.26.

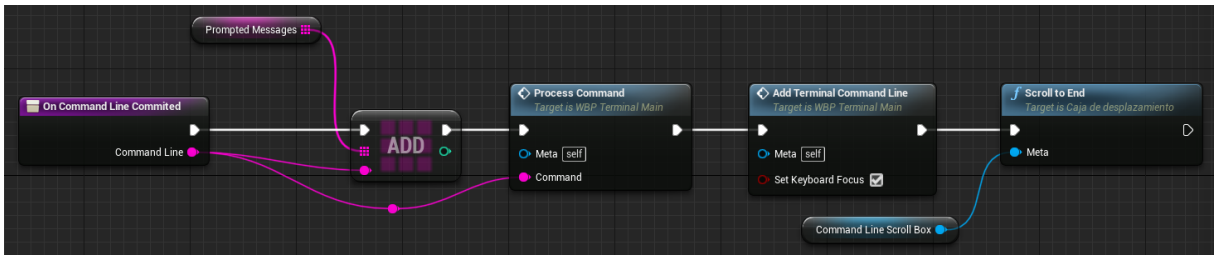


Figura 5.25: Función *OnCommandLineCommited*

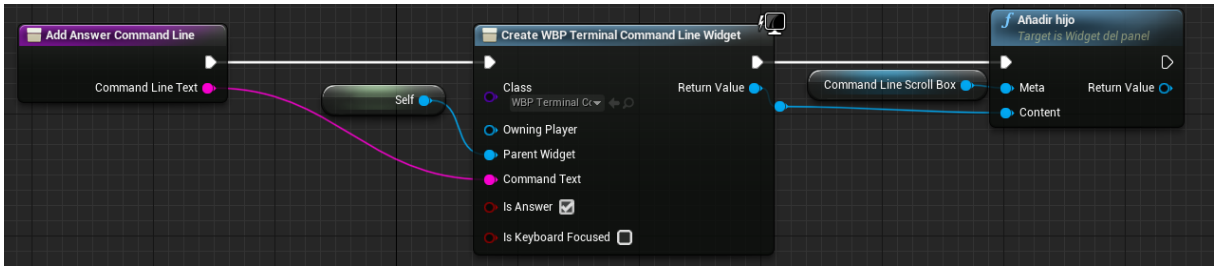


Figura 5.26: Función *AddCommandLine*

Como se ha explicado, la función *OnCommandLineCommited* procesa la información que envía el *widget WBP_TerminalCommandLine*, se llama a la función *ProcessCommand* (ver Figuras 5.27 y 5.28). En un primer momento, esta función limpia el *scrollbox* de cualquier comando anterior y se sitúa al final del mismo. A continuación, comprueba la carpeta en la que esta situado el terminal y, en base a eso,se llama a la función específica que procesa los comandos en base a la carpeta.

5.5.5.1 Procesos en la primera carpeta

Si la terminal se encuentra en la primera carpeta, es decir, en la carpeta principal, se llama a la función *ProcessFolderCommand*, el código de esta carpeta se puede ver en la Figura 5.29. Esta función comprueba si el comando enviado por el jugador es valido. Para esta carpeta hay seis comandos válidos. El primer comando que comprueba es el comando de ayuda, si se cumple esta condición se llama a la función *PrintHelpCommand*. Esta comprobación y la lógica de esta parte es común a todas los procesos de cada carpeta. La función de *PrintHelpCommand* Comprueba en que carpeta del terminal esta situado el jugador e imprime los comandos de cada carpeta con su respectiva descripción.

El siguiente comando es *Status* este comando imprime por pantalla el estado de configuración en el que se encuentra el terminal. Si el jugador introduce este comando se llama a la función *printStatus*. Esta función consta de cuatro partes diferenciadas, la primera parte se encarga de imprimir los valores establecidos por el jugador de la IP y la máscara. El detalle de este código se puede ver en las Figuras

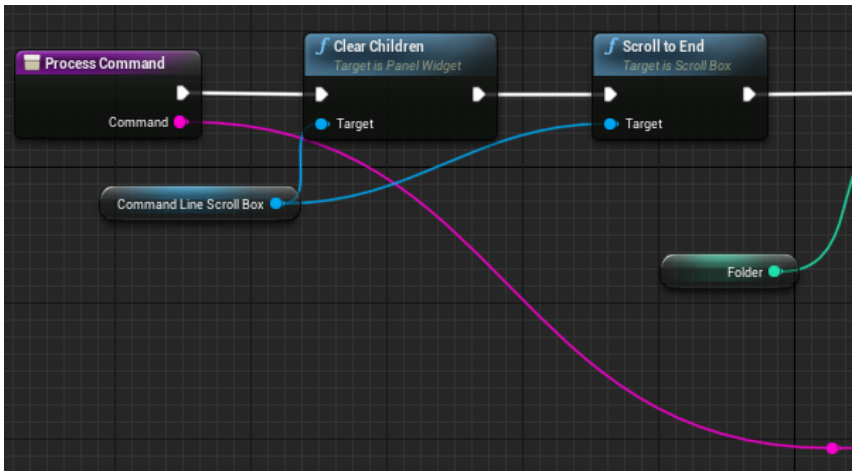


Figura 5.27: Función *ProcessCommand* (parte A)

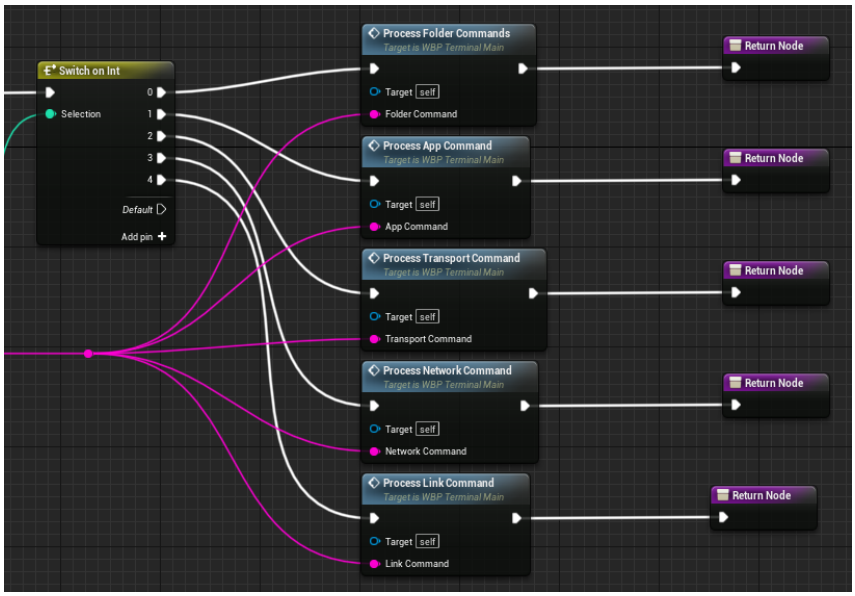


Figura 5.28: Función *ProcessCommand* (parte B)

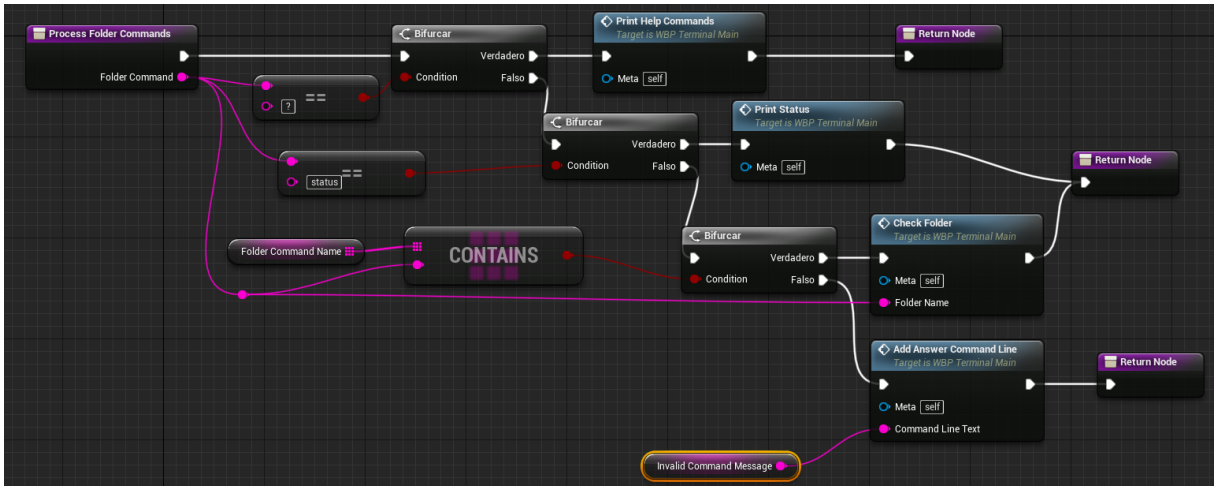


Figura 5.29: Función *ProcessFolderCommand*

5.30 y 5.31. Esta parte se encarga de comprobar si se ha realizado algún tipo de configuración en la IP y la máscara, y ambos valores se encuentran vacíos significa que no se han configurado todavía y se imprime el mensaje *"not specified"*.

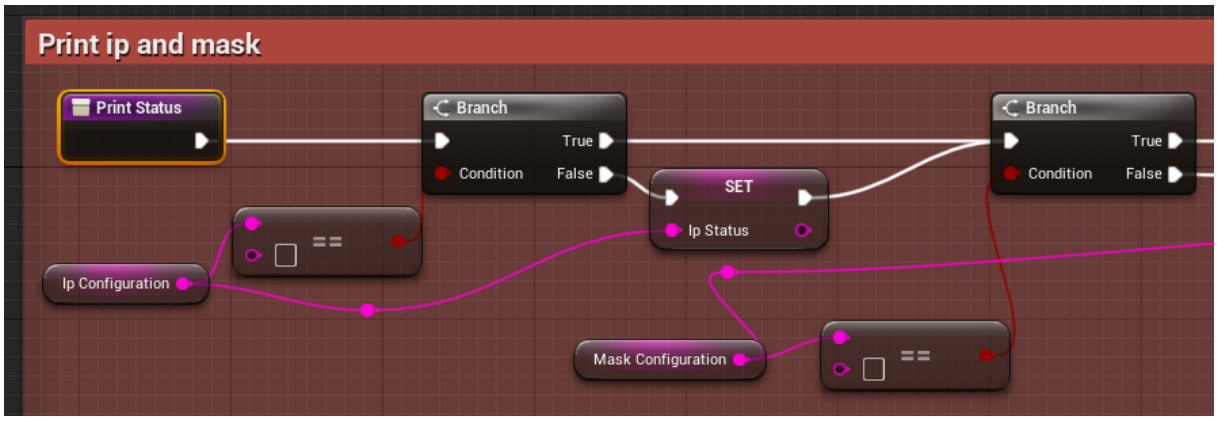


Figura 5.30: Detalle para imprimir ip y máscara (parte A)

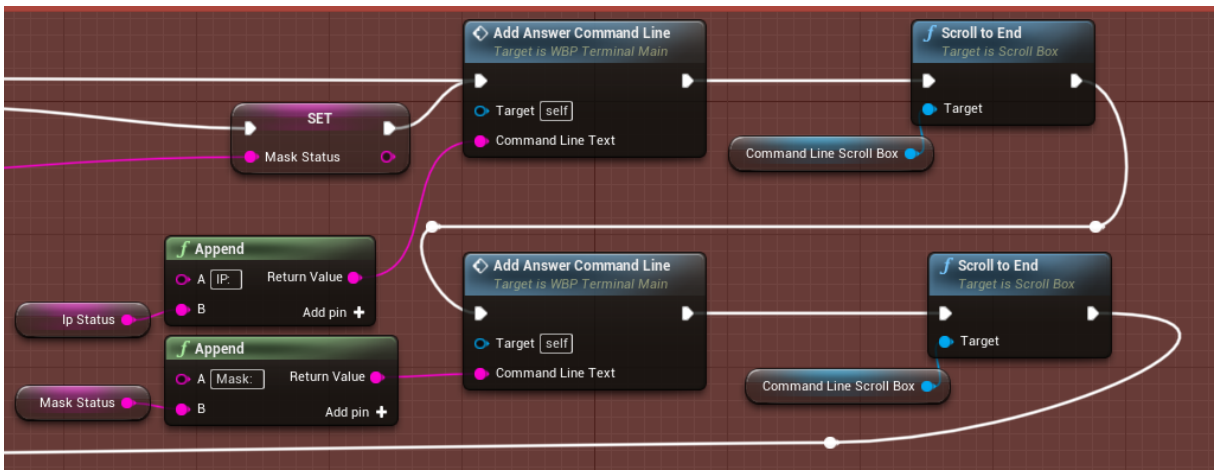


Figura 5.31: Detalle para imprimir ip y máscara (parte B)

La siguiente comprobación que realiza es la configuración de la capa de transporte. Debido a las especificaciones de nuestro sistema, este valor solo puede ser TCP o UDP. Al igual que ocurre con la IP y la máscara, se comprueba si se ha asignado algún valor o no. En el caso de que se haya asignado, se indicará por pantalla cual de las dos opciones se ha elegido. Se puede ver el código de la configuración en la Figura 5.32.

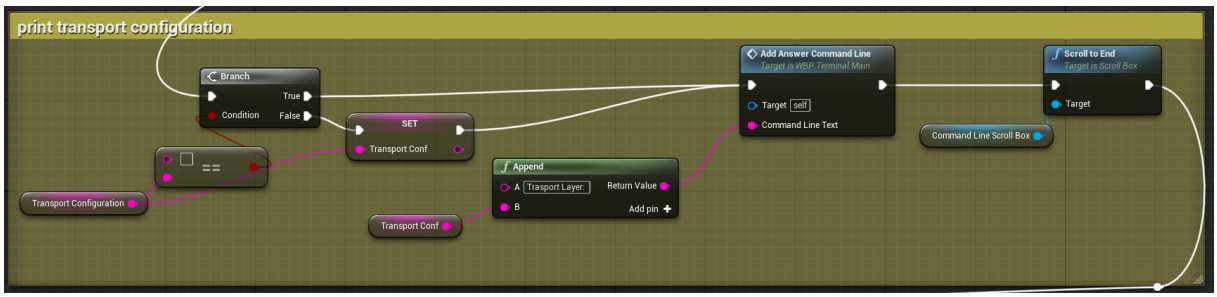


Figura 5.32: Detalle para imprimir configuración transporte

Una vez se ha comprobado la configuración de la IP la máscara y la capa de transporte, se comprueba la configuración de la capa de enlace. Al igual que ocurre con la capa de transporte, la capa *link* solo puede ser Ethernet o *wifi*. El código de esta parte se puede ver en la Figura 5.33.

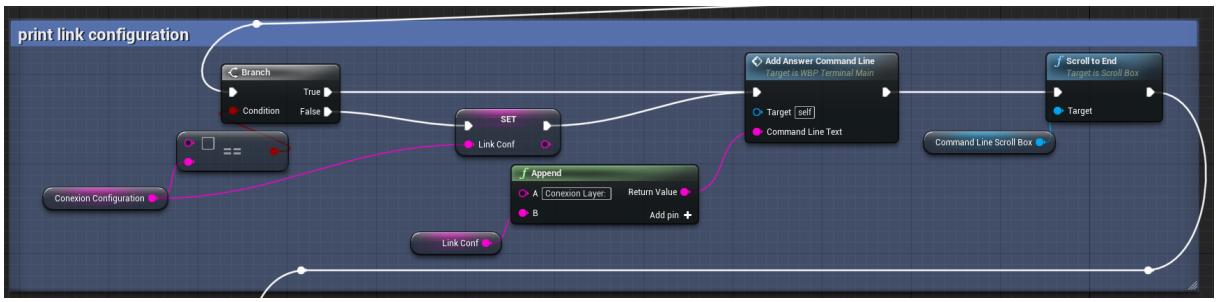


Figura 5.33: Detalle para imprimir configuración link

Por último, la ultima comprobación que se realiza es la MAC. La lógica de funcionamiento sigue los mismos pasos que lo que se ha explicado anteriormente. El código de esta parte puede ver en la Figura 5.34.

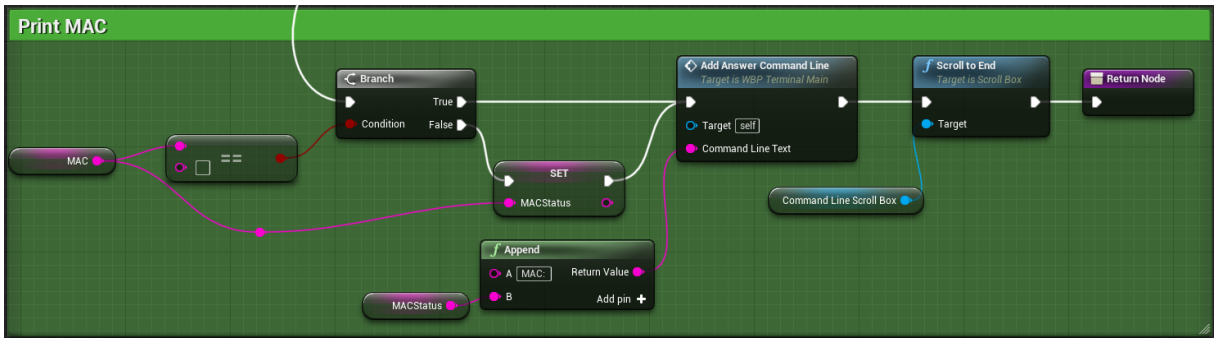


Figura 5.34: Detalle para imprimir MAC

5.5.5.2 Procesos en la carpeta APP

Para procesar los comandos de la carpeta de APP se tiene la función *ProcessAPPComand*, como se ha explicado antes, el primer comando a procesar es el de ayuda, cuyo código sigue la misma lógica que el de la carpeta principal. Además de este comando, esta carpeta, al igual que ocurre con las carpetas de *link*, *transport* y *network*, poseen un comando que retrocede hasta la carpeta principal, el código de esta lógica se puede ver en la Figura 5.35.

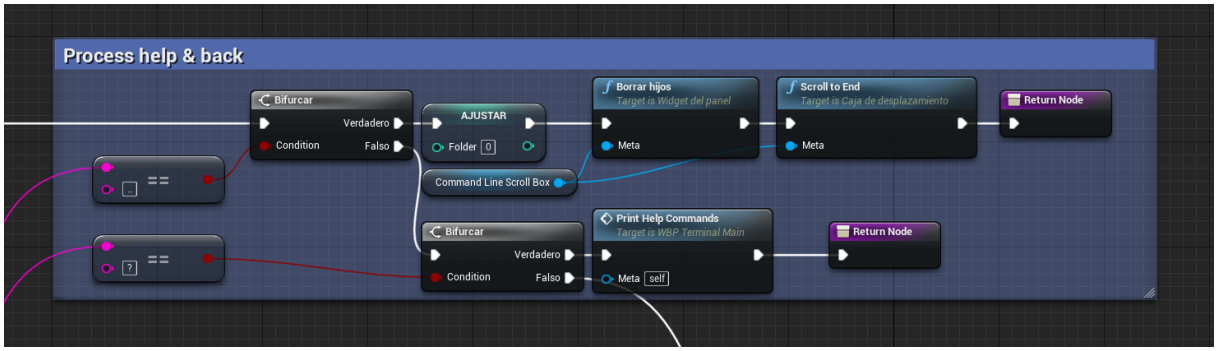


Figura 5.35: Codigo procesar comandos en app

El siguiente comando a procesar en *DNS "url"* este comando devuelve la información de la IP y la marcará de la url especificada. Si se detecta que el jugador ha introducido este comando, este proceso obtiene la IP y la máscara de la url y la imprime por pantalla. El código de esta parte puede verse en la Figura 5.36.

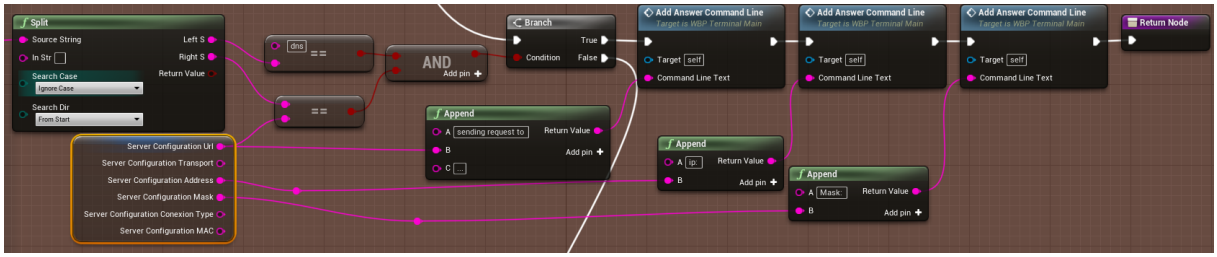


Figura 5.36: Código comando dns

A continuación, se comprueba si el comando introducido es *http "url"*. Este comando (ver Figuras 5.37 y 5.38) es el encargado de realizar la conexión con el servidor, siempre y cuando la configuración se haya realizado correctamente. Para ello si se ha introducido el comando, se llama a la función *checkConexion*, que devuelve un valor booleano indicando si la configuración es correcta o no. Si se cumple la condición, se llama al evento del terminal físico, *ServerConected*. Un evento [74] es un nodo específico que permite activación de funciones desde otras clases y actores.

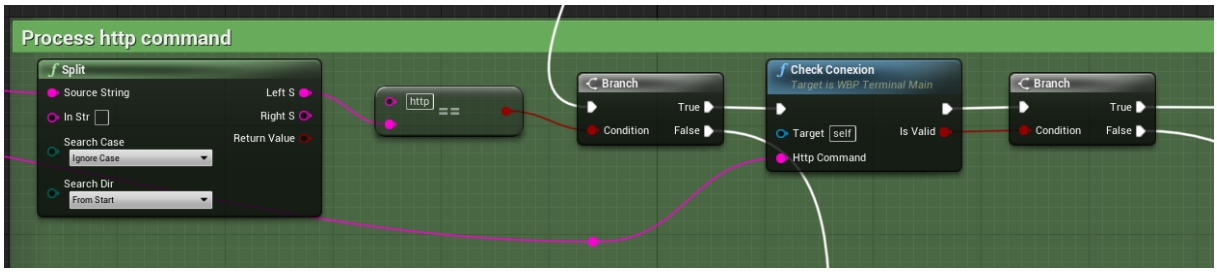


Figura 5.37: Código comando http (parte A)

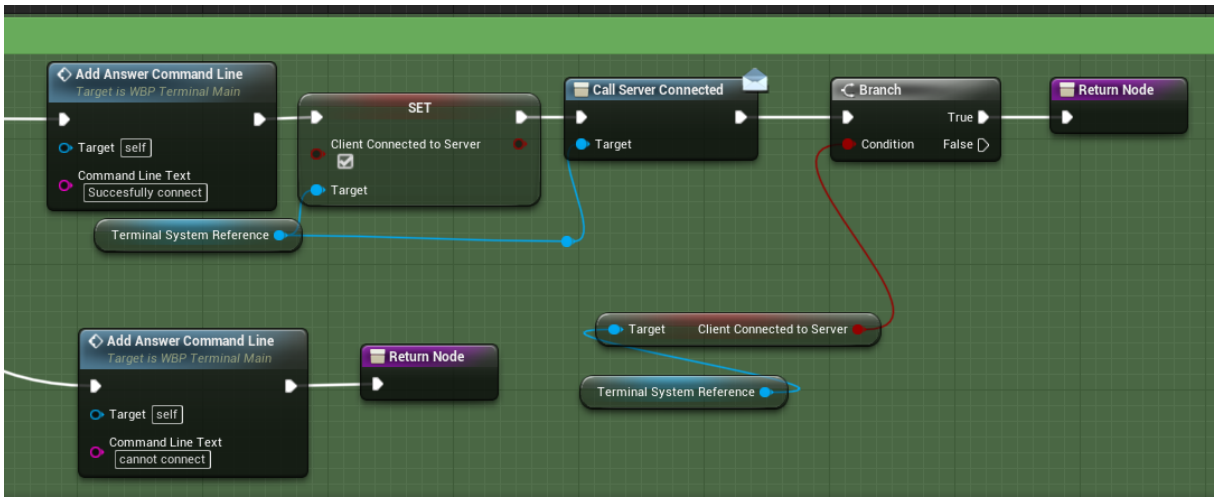


Figura 5.38: Código comando http (parte B)

Por último, si el comando introducido no coincide con ninguno de los comando válidos, se imprime por pantalla un mensaje indicándole al jugador que el comando no es válido y se le indica qué introducir

para saber qué comandos son válidos para esa carpeta. Esta última comprobación es común para todas las carpetas (ver Figura 5.39).

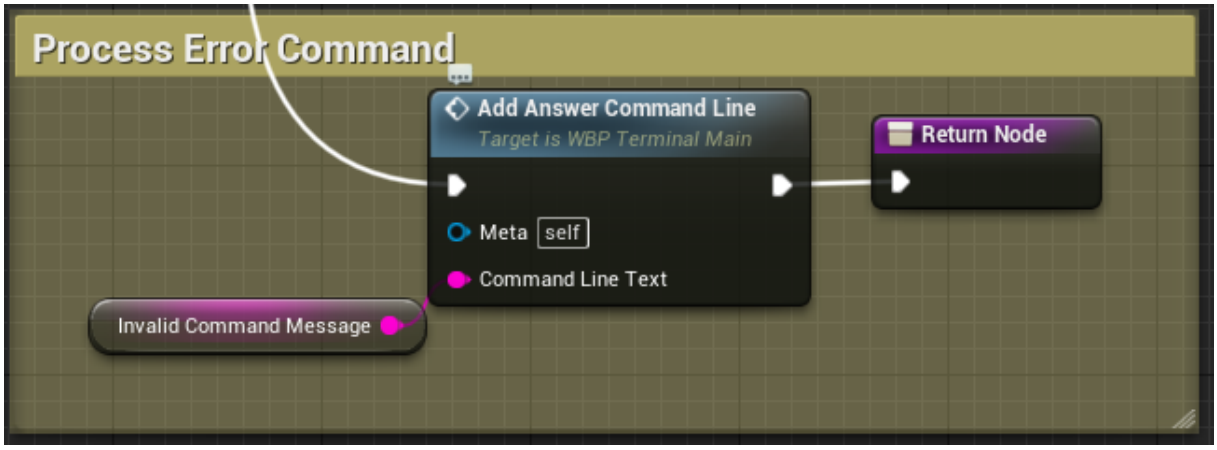


Figura 5.39: Código comando http

5.5.5.3 Procesos en la carpeta TRANSPORT

Los primeros comandos a procesar con el de ayuda y el de retroceder a la carpeta principal. Además de estos comandos comunes, en esta carpeta se procesan los comandos de tcp y udp. El proceso es bastante sencillo. Si se verifica que el jugador ha introducido uno de los dos comandos, se guarda ese comando en la variable de configuración de la capa de transporte, tal y como se ve en la Figura 5.40.

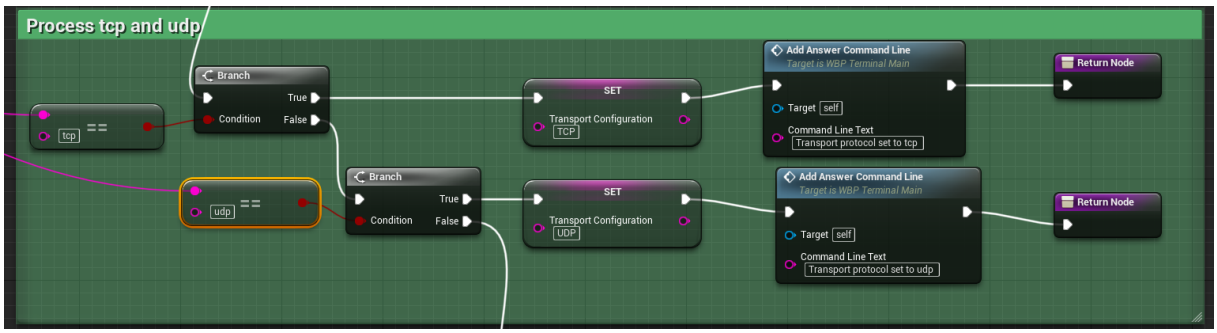


Figura 5.40: Comando tcp y udp

5.5.5.4 Procesos en la carpeta LINK

Al igual que ocurre en la carpeta de *transport*, si se verifica que se ha introducido los comandos de wifi o Ethernet se guarda en la configuración de la capa *link*. El código se puede ver en la Figura 5.41.

El otro comando que se comprueba en esta carpeta es el comando *arp "url"*. Este comando muestra y almacena en la configuración la MAC de la URL especificada tal y como se muestra en la Figura 5.42

5.5.5.5 Procesos en la carpeta NETWORK

Esta carpeta permite establecer los valores de IP y MAC del terminal del jugador, si se detecta que el comando introducido de IP o MAC, se almacena la IP o MAC que da el jugador en la variable de configuración correspondiente, esta función no comprueba si el valor es correcto o no, simplemente

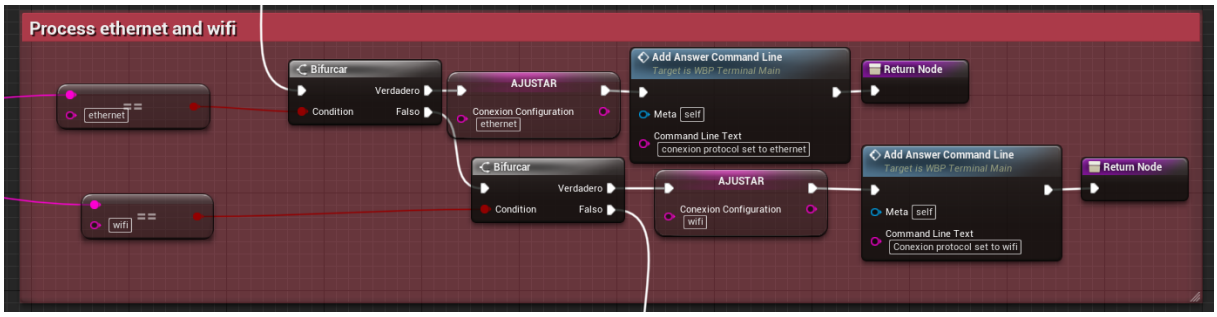


Figura 5.41: Comandos wifi y ethernet

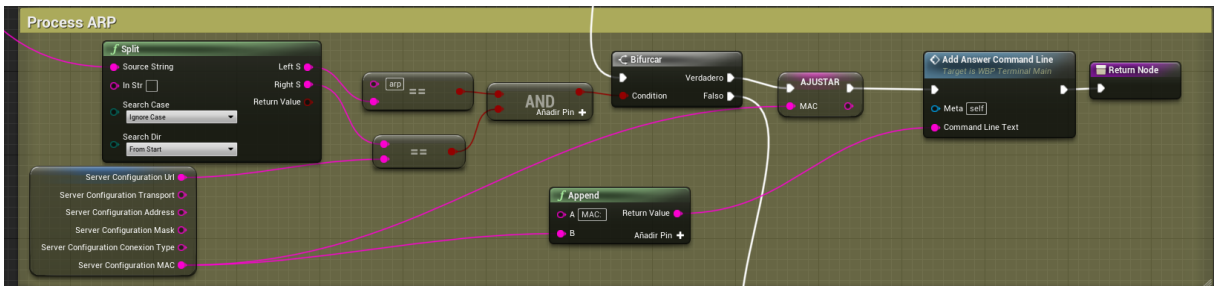


Figura 5.42: Comando arp

almacena el valor ya que el encargado de verificar si el valor es adecuado es el comando http. El código de esta función se puede ver en la Figura 5.43

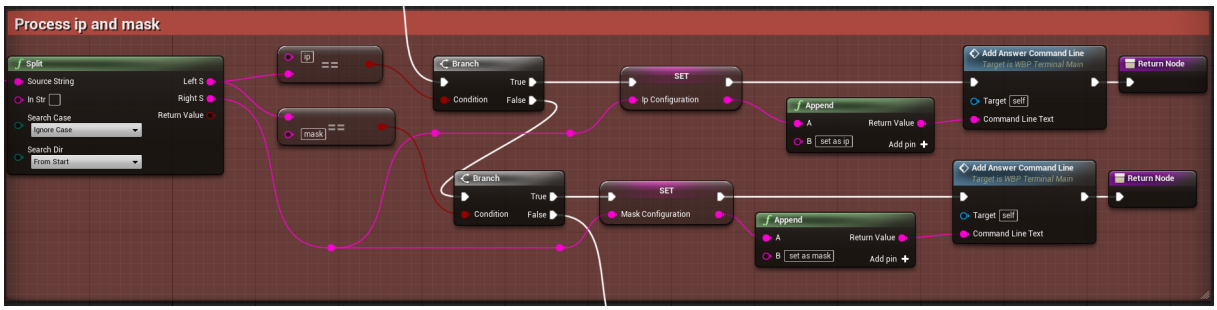


Figura 5.43: Comandos ip y mask en *network*

5.6 Otras implementaciones adicionales

En este apartado vamos a hablar del sistema de misiones y de la interfaz de usuario principal. Estas dos implementaciones no son estrictamente necesarias para el correcto funcionamiento del juego, pero permiten al jugador obtener información adicional para comprender qué debe hacer y cómo moverse por el espacio, así como cual es el objetivo final del juego.

Inicialmente vamos a hablar de la interfaz del jugador. Nada más iniciar el juego, saldrá por pantalla un texto que explica brevemente que el jugador se encuentra en una nave y que debe recolectar las tres baterías para acceder a la sala de control y conectarse con un servidor. Este texto desaparece tras un tiempo y se inicializa cada vez que el jugador reinicia el juego. En la Figura 5.44 se puede ver el código que se ejecuta cada vez que se inicializa. Este código inhabilita el *input* del jugador hasta que termina la animación y el tiempo establecido y una vez terminado devuelve el control al jugador.

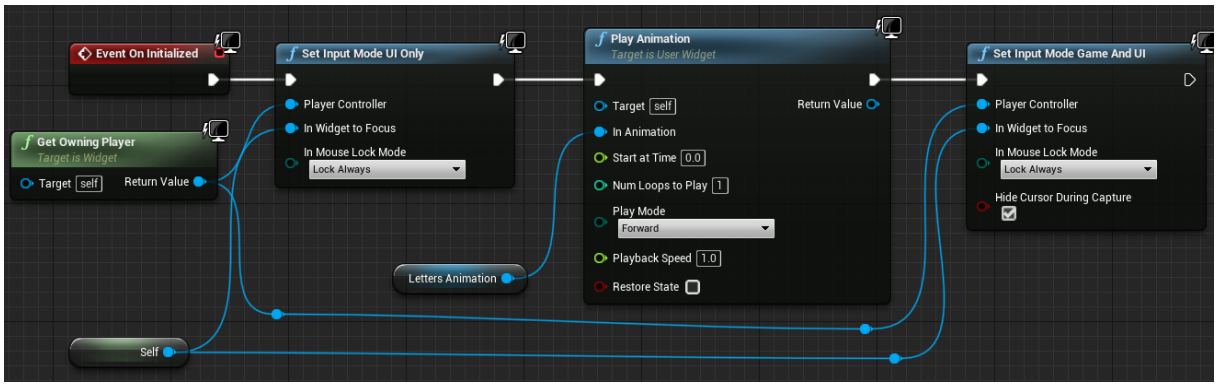


Figura 5.44: Implementación pantalla inicio

A continuación, tenemos el menú de pausa. El jugador puede acceder a él pulsando la tecla de ”tab” o escape. Este menú de pausa muestra un listado de misiones que el jugador debe ir completando. Además, en la parte derecha de la pantalla se muestran las teclas básicas de movimiento, así como dos botones para salir y reiniciar el juego. Esto se puede ver en la Figura 5.45.

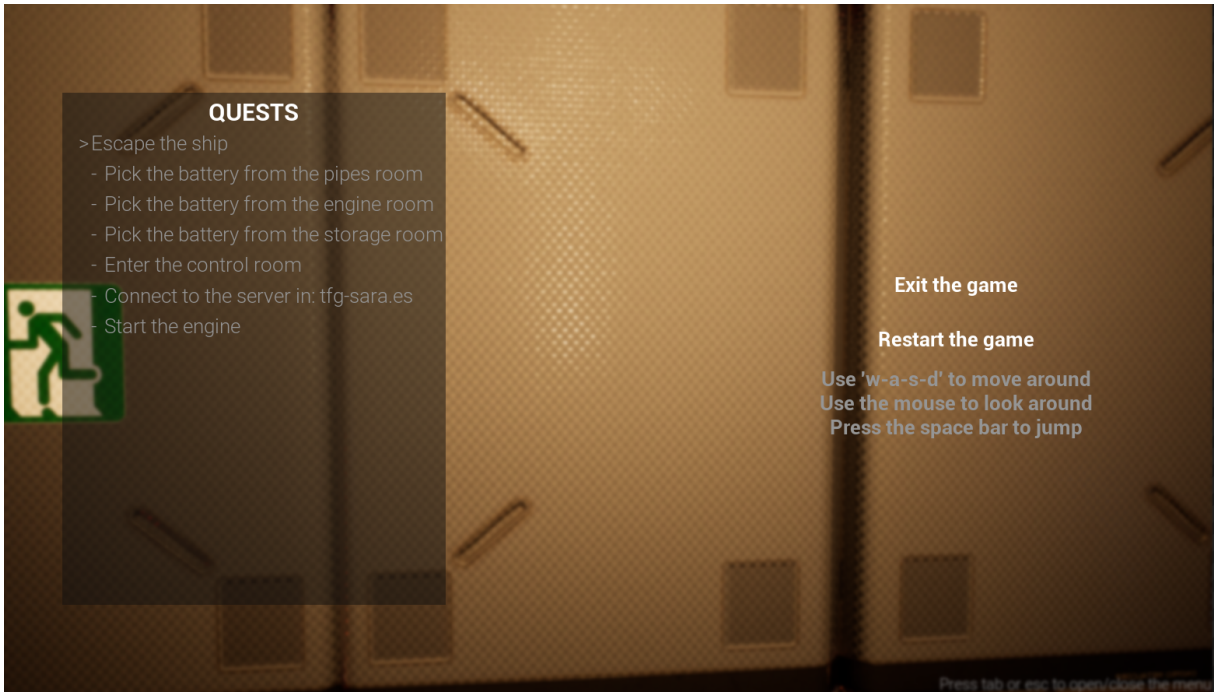


Figura 5.45: Menú de pausa

Por ultimo, tenemos la pantalla de juego final, esta pantalla se activa si el jugado ha pulsado el botón de arranque y se ha conseguido conectar con el servidor. Para hacerlo, cada vez que el jugador presiona el botón de arranque, activa un evento llamado *startEngine* (ver Figura 5.46), este evento se llama en el *Blueprint* del nivel [75]. Al iniciar el nivel se enlazan dos eventos a la lógica del nivel. Estos eventos son *startEngine* y *restartLevel*. Esto se puede ver en la Figura 5.47.

En el momento en el que se llama al evento, se activa la función *WinScreen*. Lo primero que hace esta función es comprobar si el jugador se encuentra dentro del rango del botón, si se cumple se comprueba si el terminal ha realizado la conexión correctamente. En caso afirmativo añade a la pantalla el *widget* de que el jugador ha ganado. Esta pantalla tiene dos botones que, al igual que ocurre con el menú de pausa, sirven para salir del juego y para reiniciar el nivel.

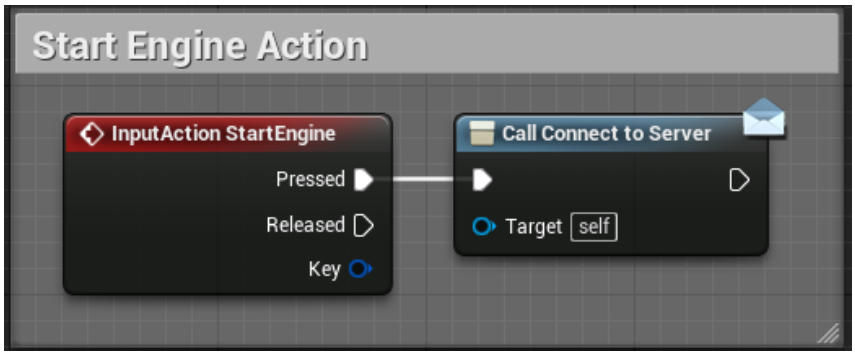


Figura 5.46: Llamada del evento *startEngine*

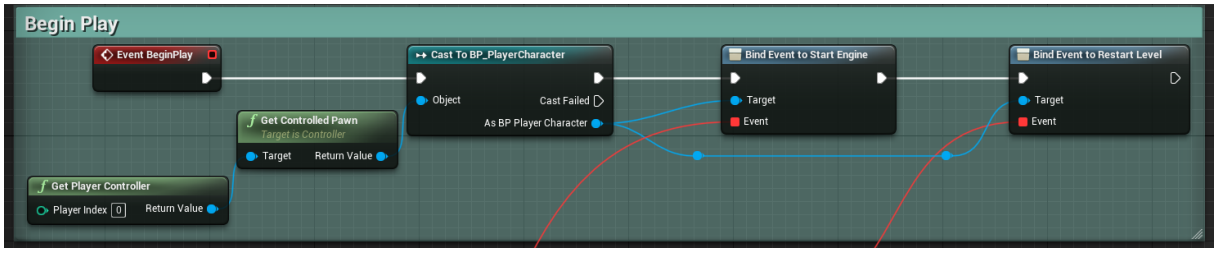


Figura 5.47: Vinculación de eventos al inicio del nivel

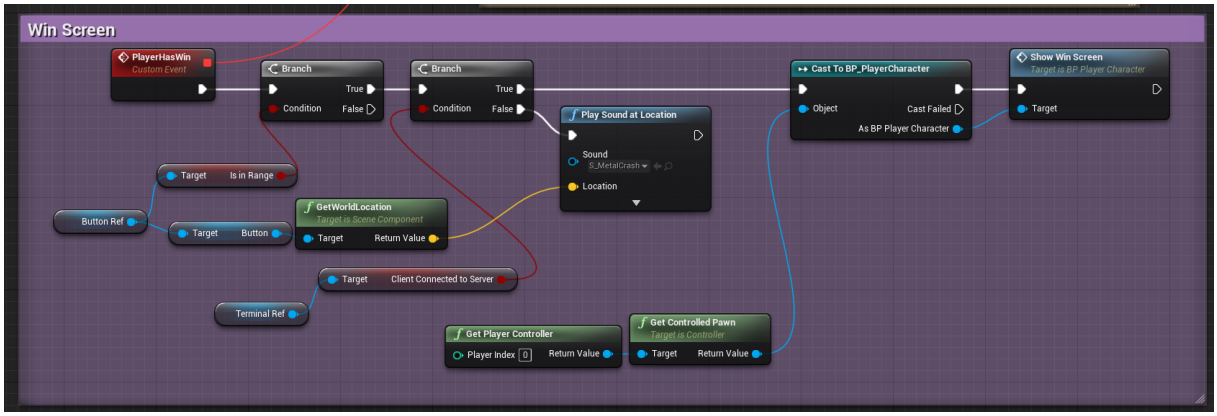


Figura 5.48: Activación evento *startEngine*

5.7 Últimos pasos: compilación del proyecto

Una vez hemos terminado la programación de la lógica y se han realizado todas las pruebas necesarias para comprobar que funciona, es hora de pasar al empaquetado final del proyecto [76]. Durante este empaquetado lo primero en procesarse es el código fuente del proyecto, una vez se ha procesado esa información, se para a la conversión de los contenidos necesarios para la plataforma objetivo. Una vez terminado el proceso se obtendrá un conjunto de archivos junto a un ejecutable.

Para acceder a las opciones de compilación debemos ir a la pestaña de archivos y dirigirnos a la opción de *package project* (ver Figura 5.49). Se abrirá un desplegable en el cual se puede seleccionar la plataforma para la que queremos obtener el proyecto, sin embargo, hay ciertas limitaciones ya que a la hora de crear el proyecto la primera vez debemos seleccionar las plataformas objetivo. En nuestro caso particular al haber seleccionado que nuestra plataforma era el ordenador de sobremesa o portátil (*PC*), sólo podremos realizar el empaquetado para las plataformas de linux, MAC OS y Windows.

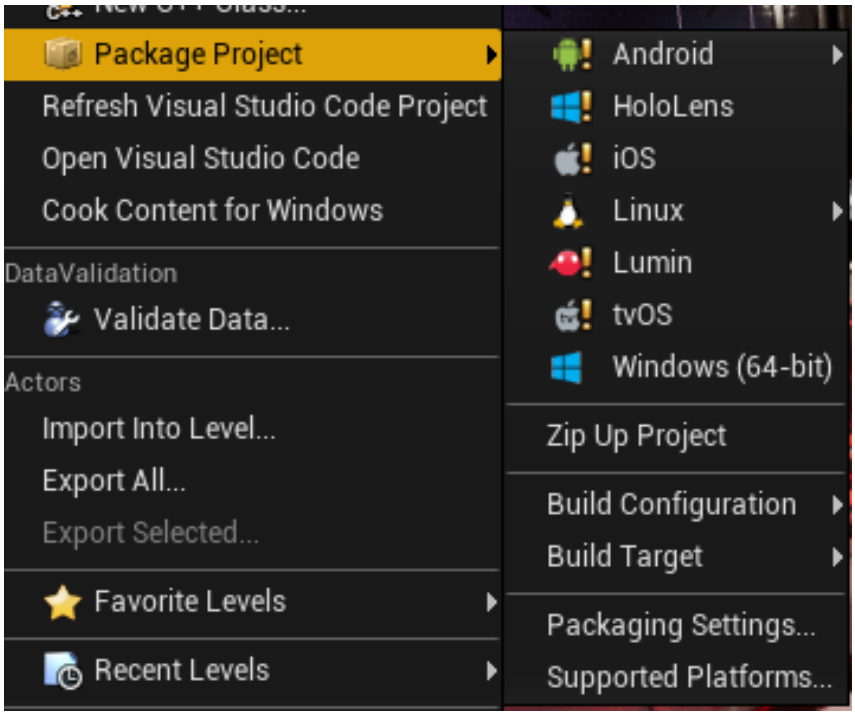


Figura 5.49: Opciones de empaquetado

Antes de realizar el empaquetado final es recomendable asegurarse de que el mapa por defecto es el que queremos, para hacerlo debemos ir a los ajustes de nuestro proyecto y en mapas y modos asegurarnos de que en la opción de *game default maps* se ha seleccionado el mapa que queremos. En la Figura 5.50 se puede ver la configuración para este TFG.

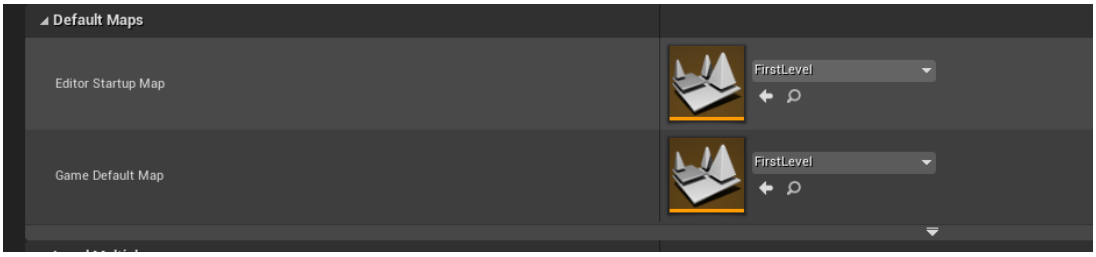


Figura 5.50: Elección de mapa en empaquetado

Otra consideración a tener el cuenta son los mapas que se van a añadir al empaquetado. Si especificamos los mapas que se van a usar, solo se procesarán esos mapas y no todos los *assets* del proyecto. Esto tiene la ventaja de que se obtiene un producto mucho más ligero en el caso de que se hayan añadido elementos que por diferentes motivos al final no se han llegado a usar. Por otro lado, hay que tener cuidado de añadir todos los mapas que si se van a usar ya que si se olvida uno de los mapas puede dar errores. En la Figura 5.51 se puede ver los mapas específicos para este TFG. Esta opción se encuentra en la configuración del proyecto, dentro de la pestaña de empaquetado (*packing*).

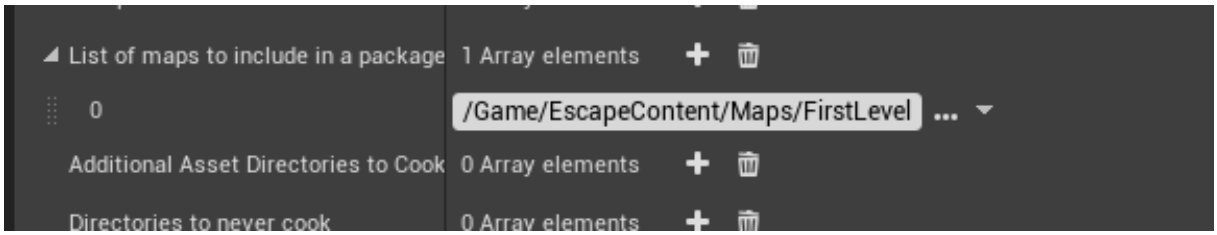


Figura 5.51: Ajustes de empaquetado

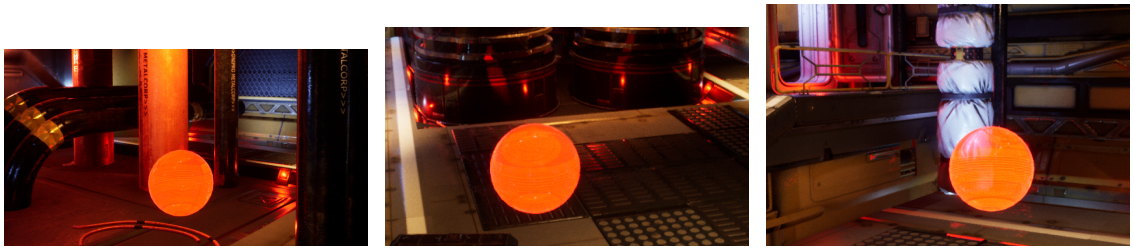
Capítulo 6

Evaluación

A lo largo de este proyecto se ha investigado y desarrollado una aplicación usando como herramienta el motor gráfico Unreal Engine. Para poder probar su aplicación como método de enseñanza, el juego se ha probado con varias personas con diferentes niveles de conocimiento sobre la aplicación de redes. Estas pruebas se han realizado para verificar que el juego es lo suficientemente intuitivo en cuanto al movimiento y controles básicos, es decir, acceso al menú y comprensión básica de los controles y objetivos del juego y que el juego es lo suficientemente intuitivo como para que una persona, con los conocimientos necesarios en redes, sea capaz de terminar el juego sin ayuda.

El proceso que debe seguir el jugador para completar el juego es el siguiente:

- Lo primero que debe hacer el jugador es coleccionar las tres baterías que se encuentran repartidas por el mapa: La primera en la sala de tuberías, la segunda en la sala de motores y la última en la cubierta de carga. Las localizaciones se pueden ver en las figura 6.1



(a) Batería sala de tuberías (b) Batería sala de motores (c) Batería cubierta de carga

Figura 6.1: Localización de las baterías

- Una vez el jugador haya recolectado todas las baterías debe dirigirse a la puerta de la sala de control la cuál se abrirá automáticamente, dando acceso al terminal.
- Una vez se tiene acceso se puede pasar a la configuración. El orden de configuración es irrelevantes, pero deben realizarse todos los pasos para que la configuración sea correcta.
- Dentro de la carpeta app obtenemos la información de la IP y la máscara introduciendo la URL del servido (tfg-sara.es), esto se ve en la Figura 6.2.
- Una vez se sabe la IP y la máscara correspondiente con el servidor, se puede configurar la IP y la máscara del cliente. Esto se hace desde la carpeta de *network*. La configuración de la IP y la máscara se ve en las Figuras 6.3 y 6.4.

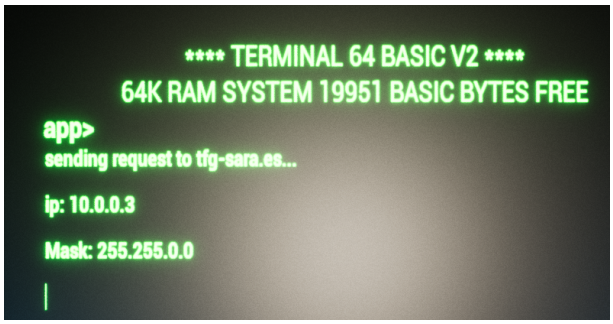
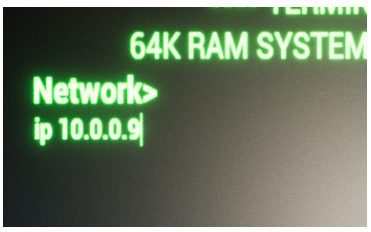


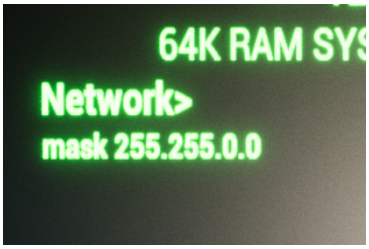
Figura 6.2: Respuesta al comando dns tfg-sara.es



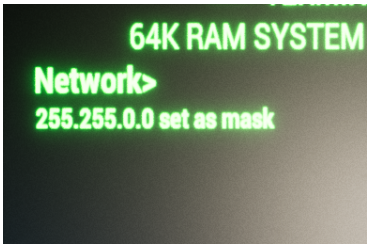
(a) Comando de configuración IP cliente

(b) Respuesta del comando de configuración IP del cliente

Figura 6.3: Configuración IP



(a) Comando de configuración de máscara cliente



(b) Respuesta del comando de configuración de máscara del cliente

Figura 6.4: Configuración de la máscara del cliente

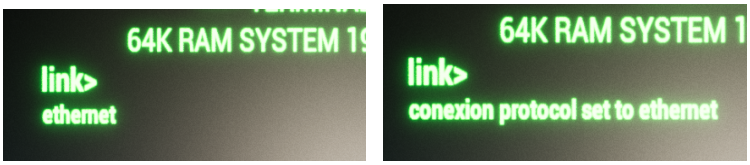
- La siguiente configuración a realizar es establecer el protocolo de la capa de transporte, para ello basta con introducir, para este caso concreto, el comando tcp tal y como se muestra en la Figura 6.5.

(a) Comando de configuración de tcp

(b) Respuesta del comando de configuración tcp

Figura 6.5: Configuración del protocolo de transporte

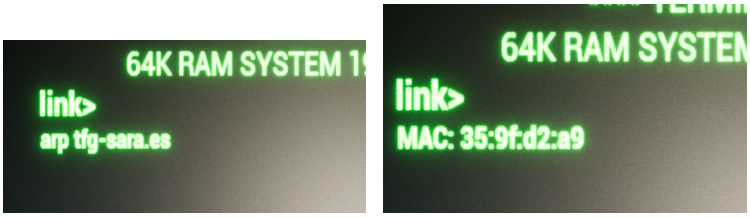
- A continuación, se pasa a las configuraciones en la carpeta de link. En este caso debemos establecer el tipo de conexión (wifi o ethernet) y se debe almacenar la dirección MAC del servidor usando el comando arp y la URL del servidor. Los comandos y las respuestas se pueden ver en las Figuras 6.6 y 6.7.



(a) Comando de configuración ethernet

(b) Respuesta del comando de configuración ethernet

Figura 6.6: Configuración ethernet



(a) Comando de configuración de máscara cliente

(b) Respuesta del comando de configuración de máscara del cliente

Figura 6.7: Configuración de la máscara del cliente

- Una vez realizadas todas las configuraciones se puede comprobar el estado de las mismas usando el comando status en la carpeta principal. Esto es completamente opcional y no afecta al resultado final. Para este caso concreto la respuesta de este comando a la configuración que se ha realizado se puede ver en la Figura 6.8
- Como último paso antes de salir del terminal se debe realizar la conexión, para ello, se debe introducir el comando http y la url dentro de la carpeta de app, si la configuración es correcta saldrá un mensaje indicando que la conexión se ha realizado, esto se ve en la Figura 6.9

Como el objetivo de este trabajo es la aplicación en la enseñanza se ha probado el juego con un total de cinco personas, dos de esas personas no poseían conocimiento sobre redes. Las otras tres personas, han cursado estudios superiores y sí poseían ese conocimiento. El tiempo medio de resolución ha sido cerca

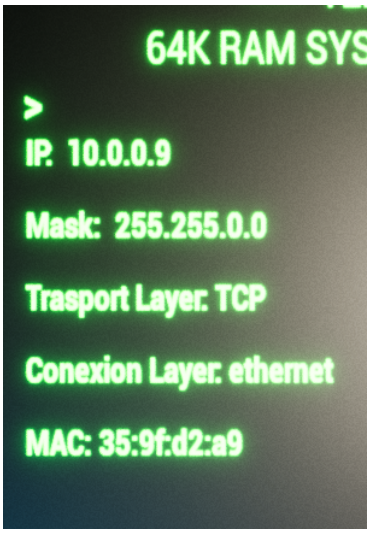
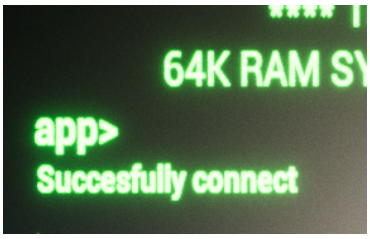


Figura 6.8: Resultado del comando status

(a) Comando http



(b) Respuesta de la conexión con el servidor

Figura 6.9: Conexión con el servidor

de los quince minutos. Sobre la intuición de los controles, no hubo ningún problema y todas las personas consiguieron llegar a la sala de control sin ningún tipo de ayuda. Una vez dentro de la sala de control, el grupo con conocimiento sobre redes consiguió resolver el problema sin ningún tipo de pista o ayuda externa. Para el caso de las personas sin conocimiento de redes, se les dio pistas externas al juego, para que pudieran superarlo.

Respecto a la realización del propio proyecto, considero que Unreal posee suficientes tutoriales y materiales para trabajar con él sin demasiados problemas, siempre y cuando se tengan claros los conceptos básicos. Sin embargo, para su realización solo se ha usado una pequeña porción de su funcionalidad, por lo que el potencial que adquiere es mucho mayor. Por ejemplo, no se ha tratado con las herramientas para el manejo de materiales y texturas. No se han usado en profundidad herramientas de *debug* y de rendimiento que proporciona el motor y tampoco se ha trabajado con herramientas de creación y edición de vídeo dentro del juego (cinemáticas).

Capítulo 7

Conclusiones y trabajo futuro

Este proyecto ha tenido una duración aproximada de cuatro meses, sin contar los meses dedicados al aprendizaje del motor. Durante ese tiempo, una sola persona ha conseguido desarrollar un producto final, sin demasiados problemas a nivel técnico. Unreal es un motor con una gran capacidad de desarrollo, que permite y facilita la creación de juegos, haciendo que la mayor complejidad resida en el diseño del juego y no los posible problemas técnicos que se puedan encontrar por el camino. La curva de aprendizaje es lenta y puede llegar a ser abrumadora, pero se consiguen resultados relativamente rápido.

A un nivel más específico del proyecto, considero que hay muchas mejoras que se pueden implementar. Por ejemplo, se podría añadir más complejidad al juego añadiendo variabilidad a las conexiones del servidor. El mapa podría ampliarse y que la apertura de la sala de control se realice con más terminales mediante el uso de otros comandos. Las posibilidades son infinitas. Respecto a las aplicaciones para la enseñanza, considero que el proyecto puede servir como refuerzo a los conocimientos impartidos en el aula, siempre y cuando el diseño se adecue a los conocimientos que se quieran impartir. Sin embargo, durante este proceso de desarrollo me he encontrado con una ventaja adicional que no había tenido en consideración en un primer momento. El uso de la programación por *scripts*, y la forma visual que tiene Unreal de trabajar con las clases ha mejorado mi comprensión de la programación orientada a objetos. Además, al tratarse de algo tan visual e interactivo ayuda a ver los errores con más facilidad y creo que podría aprovecharse ese potencial. Además, el obtener resultados rápidos y tan llamativos como los que se consiguen con Unreal, ayuda mucho a la motivación y a seguir desarrollando y programando con él.

Sin embargo, Unreal no es el único motor en el mercado, cada año salen nuevos motores y librerías relacionadas con el desarrollo de videojuegos y con la computación de gráficos tridimensionales. En junio de 2021, Epic Games publicó una demostración técnica de la nueva versión de Unreal 5, con nueva tecnología de renderizado y procesado de la iluminación [77]. Ese mismo mes, Amazon [78] liberó el código de su motor gráfico, basado en en el CryEngine [79], y lo hizo gratuito [80].

El campo de estudio de la computación, a cualquier nivel, sigue una evolución exponencial, al igual que ocurre con el desarrollo de motores gráficos. Hace apenas diez años, era impensable que una sola persona pudiese desarrollar un juego completamente funcional en apenas cuatro meses. Actualmente esto es una realidad y hay que aprovechar esa oportunidad.

Bibliografía

[1] EDUCACIÓN 3.0. *¿Qué es la gamificación y cuáles son sus objetivos?* URL: https://www.educaciontrespuntocero.com/noticias/gamificacion-que-es-objetivos/. (Fecha de consulta: 15.06.2020).

[2] *Kahoot.* URL: https://kahoot.com/. (Fecha de consulta: 10.7.2020).

[3] EDUCACIÓN 3.0. *27 herramientas de gamificación para clase que engancharán a tus alumnos.* URL: https://www.educaciontrespuntocero.com/recursos/herramientas-gamificacion-educacion/. (Fecha de consulta: 15.06.2020).

[4] *La gamificación en el aula: qué es y cómo aplicarla.* URL: https://www.unir.net/educacion/revista/gamificacion-en-el-aula/. (Fecha de consulta: 15.06.2020).

[5] Full Scale. *What is Game Engine?* URL: https://fullscale.io/blog/what-is-game-engine/. (Fecha de consulta: 16.06.2020).

[6] Game Designing. *The Power of Video Game Engines: Every Game Developer's (Not-So-Secret) Weapon.* URL: https://www.gamedesigning.org/career/video-game-engines/. (Fecha de consulta: 16.06.2020).

[7] Christensson P. *Middleware.* URL: https://techterms.com/definition/middleware. (Fecha de consulta: 16.06.2020).

[8] Game Designing. *The Engine Survey: Technology Results.* URL: https://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php. (Fecha de consulta: 16.06.2020).

[9] Ward J. *What is a Game Engine?* URL: https://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=2. (Fecha de consulta: 16.06.2020).

[10] Evan-Amos. *File:Atari-2600-Console.jpg.* URL: https://es.m.wikipedia.org/wiki/Archivo:Atari-2600-Console.jpg. (Fecha de consulta: 27.7.2020).

[11] Retro informática el pasado del futuro. *Historia de los videojuegos.* URL: https://www.fib.upc.edu/retro-informatica/historia/videojocs.html. (Fecha de consulta: 16.06.2020).

[12] *Historia de los videojuegos.* URL: https://en.wikipedia.org/wiki/Pinball_Construction_Set. (Fecha de consulta: 16.06.2020).

[13] Electronic Arts. *File:Pinball Construction Set Title Screenshot.png.* URL: https://en.wikipedia.org/wiki/File:Pinball_Construction_Set_Title_Screenshot.png. (Fecha de consulta: 27.7.2020).

[14] Evan-amos. *File:SNES-Mod1-Console-Set.jpg.* URL: https://commons.wikimedia.org/wiki/File:SNES-Mod1-Console-Set.jpg. (Fecha de consulta: 27.7.2020).

[15] Evan-Amos. *File:Sega-Mega-Drive-JP-Mk1-Console-Set.jpg*. URL: https://commons.wikimedia.org/wiki/File:Sega-Mega-Drive-JP-Mk1-Console-Set.jpg. (Fecha de consulta: 27.7.2020).

[16] Javier Castillo. *DOOM: Cuando Bill Gates presentó, en 1995 y escopeta en mano, el mítico juego*. URL: https://www.alfabetajuega.com/noticia/doom-cuando-bill-gates-presento-en-1995-y-escopeta-en-mano-el-mitico-juego-n-73853. (Fecha de consulta: 27.7.2020).

[17] *Epic Games*. URL: https://www.epicgames.com/store/es-ES/. (Fecha de consulta: 10.7.2020).

[18] *id Software*. URL: https://www.idsoftware.com/en-gb. (Fecha de consulta: 10.7.2020).

[19] *Rockstar Advanced Game Engine*. URL: https://gta.fandom.com/wiki/Rockstar_Advanced_Game_Engine. (Fecha de consulta: 16.06.2020).

[20] *THE SNOWDROP ENGINE*. URL: https://www.ubisoft.com/snowdrop. (Fecha de consulta: 16.06.2020).

[21] *Los 10 Mejores Motores para la Creación de Videojuegos*. URL: https://ciberninjas.com/motores-videojuegos/. (Fecha de consulta: 16.06.2020).

[22] *maya autodesk*. URL: https://www.autodesk.es/products/maya/overview. (Fecha de consulta: 11.7.2020).

[23] *Blender*. URL: https://www.blender.org/. (Fecha de consulta: 11.7.2020).

[24] *Bioshock*. URL: https://es.wikipedia.org/wiki/BioShock. (Fecha de consulta: 11.7.2020).

[25] *Deux Ex*. URL: https://es.wikipedia.org/wiki/Deus_Ex_(serie)#Deus_Ex_(2000). (Fecha de consulta: 11.7.2020).

[26] *Unity or Unreal Engine in 2020*. URL: https://devga.me/guides/unity-or-unreal-engine-in-2020/. (Fecha de consulta: 16.06.2020).

[27] Harvard L. *Game Theory: On BioShock, violence, Potempkin and propaganda*. URL: http://www.digitallydownloaded.net/2020/06/game-theory-on-bioshock-violence.html. (Fecha de consulta: 27.7.2020).

[28] Diógenes Pantarújez. *A la sombra de un juego sobrevalorado: Bioshock Infinite*. URL: https://brainstomping.com/2013/04/01/a-la-sombra-de-un-juego-sobrevalorado-bioshock-infinite/. (Fecha de consulta: 27.7.2020).

[29] *Unity*. URL: https://unity.com/es. (Fecha de consulta: 11.7.2020).

[30] *Unity VS Unreal Engine 4 | Which Engine Is Right For You?* URL: https://www.youtube.com/watch?v=M5FEsrbsb_M&t=2s. (Fecha de consulta: 16.06.2020).

[31] *Subnautica: Below Zero nos muestra su versión de PS5 en un nuevo tráiler*. URL: https://mediamaster.vandal.net/m/69961/subnautica-below-zero-20212217412326_7.jpg. (Fecha de consulta: 27.7.2020).

[32] Alnih. *Tiburón*. URL: https://seaofthieves.fandom.com/es/wiki/Tibur%5C%C3%5C%B3n?file=Tibur%5C%25C3%5C%25B3n.png. (Fecha de consulta: 27.7.2020).

[33] *GitHub*. URL: https://github.com/. (Fecha de consulta: 11.7.2020).

[34] *Choose the plan that is right for you*. URL: https://store.unity.com/compare-plans?gclid=CjwKCAjwwqaGBhBKEiwAMk-FtM-5at1_LSOEoSNjr2svFLOwmMVv2QhiGE0ZWYI_L-b9iBQmTygt8xoCXZEQAvD_BwE. (Fecha de consulta: 16.06.2020).

[35] *Unreal® Engine End User License Agreement For Creators*. URL: https://www.unrealengine.com/en-US/eula/creators. (Fecha de consulta: 16.06.2020).

[36] *Unreal® Engine End User License Agreement For Publishing*. URL: https://www.unrealengine.com/en-US/eula/publishing. (Fecha de consulta: 16.06.2020).

[37] *OTHER LICENSING OPTIONSs*. URL: https://www.unrealengine.com/en-US/custom-license. (Fecha de consulta: 16.06.2020).

[38] *Hardware and Software Specifications*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/RecommendedSpecifications/. (Fecha de consulta: 16.06.2020).

[39] *Installing Unreal Engine*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/InstallingUnrealEngine/. (Fecha de consulta: 17.06.2020).

[40] *DOWNLOAD UNREAL ENGINE*. URL: https://www.unrealengine.com/en-US/download. (Fecha de consulta: 19.06.2020).

[41] *Linux Quick Start*. URL: https://docs.unrealengine.com/4.26/en-US/SharingAndReleasing/Linux/BeginnerLinuxDeveloper/SettingUpAnUnrealWorkflow/. (Fecha de consulta: 17.06.2020).

[42] *Cloning a repository*. URL: https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository-from-github/cloning-a-repository#platform-linux. (Fecha de consulta: 17.06.2020).

[43] *Source Control*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/UI/SourceControl/. (Fecha de consulta: 17.06.2020).

[44] *Create a New Project*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/Projects/Browser/. (Fecha de consulta: 17.06.2020).

[45] *Blueprint Visual Scripting*. URL: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/. (Fecha de consulta: 20.06.2020).

[46] *Actors*. URL: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/. (Fecha de consulta: 20.06.2020).

[47] *Components*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/Components/. (Fecha de consulta: 20.06.2020).

[48] *Pawn*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Pawn/. (Fecha de consulta: 20.06.2020).

[49] *character*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Pawn/Character/. (Fecha de consulta: 20.06.2020).

[50] *PlayerController*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Controller/PlayerController/. (Fecha de consulta: 20.06.2020).

[51] *AIController*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Controller/AIController/. (Fecha de consulta: 20.06.2020).

[52] *Gameplay Framework Quick Reference*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/QuickReference/. (Fecha de consulta: 20.06.2020).

[53] *Game Mode and Game State*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/GameMode/. (Fecha de consulta: 20.06.2020).

[54] *Levels*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/Levels/. (Fecha de consulta: 20.06.2020).

[55] *World Settings*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/Levels/WorldSettings/. (Fecha de consulta: 20.06.2020).

[56] *Tools and Editors*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/ToolsAndEditors/. (Fecha de consulta: 10.7.2020).

[57] *Main Menu Bar*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelEditor/MenuBar/. (Fecha de consulta: 27.06.2020).

[58] *Level Editor Toolbar*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelEditor/Toolbar/. (Fecha de consulta: 27.06.2020).

[59] *Level Editor Modes*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelEditor/Modes/. (Fecha de consulta: 27.06.2020).

[60] *Content Browser*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/ContentBrowser/. (Fecha de consulta: 27.06.2020).

[61] *Editor Viewports*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelEditor/Viewports/. (Fecha de consulta: 27.06.2020).

[62] *World Outliner*. URL: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LevelEditor/SceneOutliner/. (Fecha de consulta: 27.06.2020).

[63] *Blueprints vs C++*. URL: http://awforsythe.com/unreal/blueprints_vs_cpp/#bp_modules. (Fecha de consulta: 11.7.2020).

[64] *Principio de Pareto*. URL: https://es.wikipedia.org/wiki/Principio_de_Pareto. (Fecha de consulta: 11.7.2020).

[65] *Collision Overview*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Physics/Collision/Overview/. (Fecha de consulta: 12.7.2020).

[66] *Input Action And Axis Mappings In UE4*. URL: https://www.unrealengine.com/en-US/blog/input-action-and-axis-mappings-in-ue4. (Fecha de consulta: 24.7.2020).

[67] *UInputComponent*. URL: https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Components/UInputComponent/. (Fecha de consulta: 24.7.2020).

[68] *Widget Blueprints*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/UMG/UserGuide/WidgetBlueprints/. (Fecha de consulta: 24.7.2020).

[69] *UMG UI Designer*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/UMG/. (Fecha de consulta: 24.7.2020).

[70] *Terminal System*. URL: https://www.unrealengine.com/marketplace/en-US/product/terminal-system. (Fecha de consulta: 27.7.2020).

[71] *USceneCaptureComponent2D*. URL: https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Components/USceneCaptureComponent2D/. (Fecha de consulta: 27.7.2020).

[72] *SMultiLineEditableText*. URL: https://docs.unrealengine.com/4.26/en-US/API/Runtime/Slate/Widgets/Text/SMultiLineEditableText/. (Fecha de consulta: 27.7.2020).

[73] *Set Input Mode Game Only*. URL: https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Input/SetInputModeGameOnly/. (Fecha de consulta: 01.8.2020).

[74] *Level Blueprint*. URL: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/LevelBlueprint/. (Fecha de consulta: 01.8.2020).

[75] *Events*. URL: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Events/. (Fecha de consulta: 10.8.2020).

[76] *Packaging Projects*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/Projects/Packaging/. (Fecha de consulta: 11.8.2020).

[77] *A first look at Unreal Engine 5*. URL: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5. (Fecha de consulta: 11.8.2020).

[78] *Amazon*. URL: https://es.wikipedia.org/wiki/Amazon. (Fecha de consulta: 11.8.2020).

[79] *cryengine*. URL: https://www.cryengine.com/. (Fecha de consulta: 11.8.2020).

[80] *Amazon libera el código de su motor de videojuegos Lumberyard: ahora es Open 3D Engine, y tiene el respaldo de la Fundación Linux*. URL: https://www.genbeta.com/desarrollo/amazon-libera-codigo-su-motor-videojuegos-lumberyard-ahora-open-3d-engine-tiene-respaldo-fundacion-linux. (Fecha de consulta: 11.8.2020).

[81] *Unreal Engine*. URL: https://www.unrealengine.com/en-US/. (Fecha de consulta: 10.7.2020).

Apéndice A

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC compatible
- Sistema operativo Windows 10 Pro
- Entorno de desarrollo Microsoft Visual Studio
- Entorno de desarrollo Visual Studio Code
- Motor Gráfico Unreal Engine [81]
- Control de versiones git [33]

Apéndice B

Presupuesto y estimación del proyecto

Antes de comenzar a realizar este proyecto, se hizo una estimación de la duración de las diferentes fases del proyecto, incluyendo la fase de aprendizaje, el diagrama de Grantt del proyecto se puede ver en la Figura B.1. Durante su ejecución, los tiempos establecidos han variado a medida que se iba desarrollando.

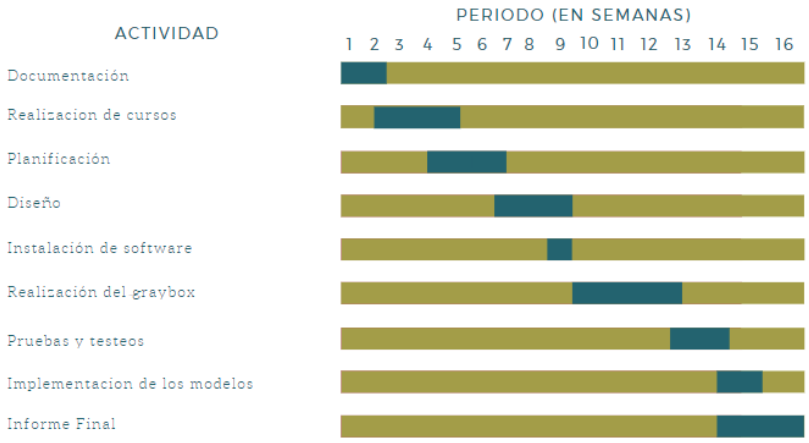


Figura B.1: Diagrama de Grantt inicial del proyecto

Por ejemplo, la fase de desarrollo y construcción del graybox se ha visto incrementada en dos semanas, pero la fase de planificación se ha reducido. Para el caso del informe final no se tuvo en cuenta la parte de investigación adicional requerida para hacer la memoria completa por lo que su duración se ha visto ampliada. En la Figura B.2 se puede ver la duración real de cada fase del proyecto.

Hablando sobre el presupuesto del proyecto hay varias consideraciones a tener el cuenta. Respecto a los gastos del proyecto, los *assets* usados se han obtenido de forma gratuita, sin embargo, si se quiere ampliar el proyecto o usar *assets* y funcionalidades de terceros hay que tener en consideración el precio. Debido a la naturaleza del proyecto, es decir, como no se tiene intención de sacarlo a la venta, no se va a obtener un beneficio económico y por tanto no hay que pagar por el uso del motor, a pesar de ello los requisitos *hardware* necesarios para usar la herramienta pueden ser limitantes ya que requiere un ordenador con una cierta capacidad y potencia.

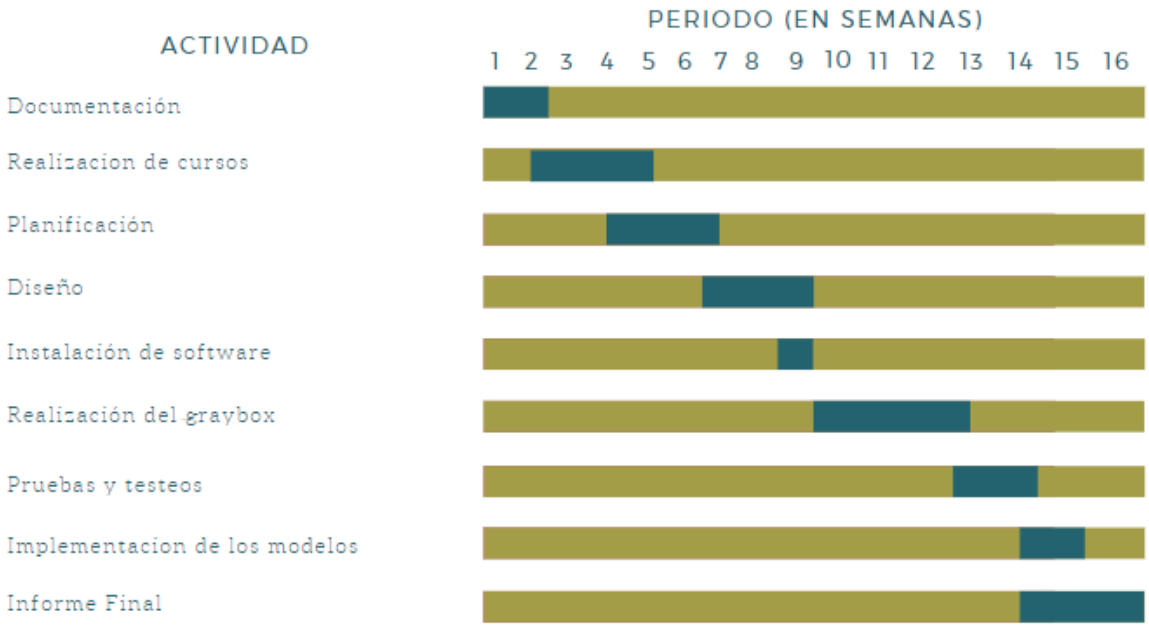


Figura B.2: Diagrama de Grantt final del proyecto

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá